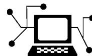
MATLAB® 7
Programming

MATLAB®

How to Contact The MathWorks



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Programming

© COPYRIGHT 1984–2007 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release R2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release R2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release R2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release R2007b)

Contents

1 Data Structures

Creating and Concatenating Matrices **1-3**
Overview **1-3**
Constructing a Simple Matrix **1-4**
Specialized Matrix Functions **1-5**
Concatenating Matrices **1-8**
Matrix Concatenation Functions **1-9**
Generating a Numeric Sequence **1-11**
Combining Unlike Data Types **1-13**

Matrix Indexing **1-18**
Accessing Single Elements **1-18**
Linear Indexing **1-19**
Functions That Control Indexing Style **1-19**
Accessing Multiple Elements **1-20**
Using Logicals in Array Indexing **1-22**
Single-Colon Indexing with Different Array Types **1-26**
Indexing on Assignment **1-26**

Getting Information About a Matrix **1-28**
Dimensions of the Matrix **1-28**
Data Types Used in the Matrix **1-29**
Data Structures Used in the Matrix **1-30**

Resizing and Reshaping Matrices **1-31**
Expanding the Size of a Matrix **1-31**
Diminishing the Size of a Matrix **1-35**
Reshaping a Matrix **1-36**
Preallocating Memory **1-38**

Shifting and Sorting Matrices **1-41**
Shift and Sort Functions **1-41**
Shifting the Location of Matrix Elements **1-41**
Sorting the Data in Each Column **1-43**
Sorting the Data in Each Row **1-43**

Sorting Row Vectors 1-44

Operating on Diagonal Matrices **1-46**
Diagonal Matrix Functions 1-46
Constructing a Matrix from a Diagonal Vector 1-46
Returning a Triangular Portion of a Matrix 1-47
Concatenating Matrices Diagonally 1-47

Empty Matrices, Scalars, and Vectors **1-48**
Overview 1-48
The Empty Matrix 1-49
Scalars 1-51
Vectors 1-52

Full and Sparse Matrices **1-54**
Overview 1-54
Sparse Matrix Functions 1-54

Multidimensional Arrays **1-56**
Overview 1-56
Creating Multidimensional Arrays 1-58
Accessing Multidimensional Array Properties 1-62
Indexing Multidimensional Arrays 1-62
Reshaping Multidimensional Arrays 1-66
Permuting Array Dimensions 1-68
Computing with Multidimensional Arrays 1-70
Organizing Data in Multidimensional Arrays 1-71
Multidimensional Cell Arrays 1-73
Multidimensional Structure Arrays 1-74

Summary of Matrix and Array Functions **1-76**

2 Data Types

Overview of MATLAB Data Types **2-3**
Fundamental Data Types 2-3
How to Use the Different Types 2-4

Numeric Types **2-6**
Overview **2-6**
Integers **2-6**
Floating-Point Numbers **2-14**
Complex Numbers **2-24**
Infinity and NaN **2-25**
Identifying Numeric Types **2-27**
Display Format for Numeric Values **2-27**
Function Summary **2-29**

Logical Types **2-33**
Overview **2-33**
Creating a Logical Array **2-33**
How Logical Arrays Are Used **2-35**
Identifying Logical Arrays **2-36**

Characters and Strings **2-37**
Overview **2-37**
Creating Character Arrays **2-37**
Cell Arrays of Strings **2-39**
Formatting Strings **2-42**
String Comparisons **2-55**
Searching and Replacing **2-58**
Converting from Numeric to String **2-59**
Converting from String to Numeric **2-61**
Function Summary **2-63**

Dates and Times **2-66**
Overview **2-66**
Types of Date Formats **2-66**
Conversions Between Date Formats **2-68**
Date String Formats **2-69**
Output Formats **2-70**
Current Date and Time **2-71**
Function Summary **2-72**

Structures **2-74**
Overview **2-74**
Building Structure Arrays **2-75**
Accessing Data in Structure Arrays **2-78**
Using Dynamic Field Names **2-80**
Finding the Size of Structure Arrays **2-81**

Adding Fields to Structures 2-82
Deleting Fields from Structures 2-83
Applying Functions and Operators 2-83
Writing Functions to Operate on Structures 2-84
Organizing Data in Structure Arrays 2-85
Nesting Structures 2-91
Function Summary 2-92

Cell Arrays 2-93
Overview 2-93
Cell Array Operators 2-94
Creating a Cell Array 2-95
Referencing Cells of a Cell Array 2-99
Deleting Cells 2-106
Reshaping Cell Arrays 2-106
Replacing Lists of Variables with Cell Arrays 2-107
Applying Functions and Operators 2-108
Organizing Data in Cell Arrays 2-109
Nesting Cell Arrays 2-110
Converting Between Cell and Numeric Arrays 2-112
Cell Arrays of Structures 2-113
Function Summary 2-114

Function Handles 2-115
Overview 2-115
Constructing and Invoking a Function Handle 2-115
Calling a Function Using Its Handle 2-116
Simple Function Handle Example 2-116

MATLAB Classes 2-117

Java Classes 2-118

3 Basic Program Components

Variables 3-2
Types of Variables 3-2
Naming Variables 3-6

Guidelines to Using Variables **3-10**
Scope of a Variable **3-10**
Lifetime of a Variable **3-12**

Keywords **3-13**

Special Values **3-14**

Operators **3-16**
Arithmetic Operators **3-16**
Relational Operators **3-17**
Logical Operators **3-19**
Operator Precedence **3-25**

MATLAB Expressions **3-27**
String Evaluation **3-27**
Shell Escape Functions **3-28**

Regular Expressions **3-30**
Overview **3-30**
MATLAB Regular Expression Functions **3-31**
Elements of an Expression **3-32**
Character Classes **3-33**
Character Representation **3-36**
Grouping Operators **3-37**
Nonmatching Operators **3-39**
Positional Operators **3-39**
Lookaround Operators **3-40**
Quantifiers **3-45**
Tokens **3-48**
Named Capture **3-53**
Conditional Expressions **3-55**
Dynamic Regular Expressions **3-57**
String Replacement **3-66**
Handling Multiple Strings **3-68**
Operator Summary **3-71**

Comma-Separated Lists **3-79**
What Is a Comma-Separated List? **3-79**
Generating a Comma-Separated List **3-79**
Assigning Output from a Comma-Separated List **3-81**

Assigning to a Comma-Separated List 3-82
How to Use the Comma-Separated Lists 3-83
Fast Fourier Transform Example 3-85

Program Control Statements 3-87
Conditional Control — if, switch 3-87
Loop Control — for, while, continue, break 3-91
Error Control — try, catch 3-94
Program Termination — return 3-95

Symbol Reference 3-96
Asterisk — * 3-96
At — @ 3-97
Colon — : 3-98
Comma — , ... 3-99
Curly Braces — { } 3-100
Dot — 3-100
Dot-Dot — 3-101
Dot-Dot-Dot (Ellipsis) — 3-101
Dot-Parentheses — .( ) 3-102
Exclamation Point — ! 3-103
Parentheses — ( ) 3-103
Percent — % 3-103
Percent-Brace — %{ %} 3-104
Semicolon — ; 3-104
Single Quotes — ' ' 3-105
Space Character 3-106
Slash and Backslash — / \ 3-106
Square Brackets — [ ] 3-107

Internal MATLAB Functions 3-108
Overview 3-108
M-File Functions 3-108
Built-In Functions 3-109
Overloaded MATLAB Functions 3-110

4 M-File Programming

Program Development 4-2
Overview 4-2
Creating a Program 4-2
Getting the Bugs Out 4-3
Cleaning Up the Program 4-4
Improving Performance 4-5
Checking It In 4-6

Working with M-Files 4-7
Overview 4-7
Types of M-Files 4-7
Basic Parts of an M-File 4-8
Creating a Simple M-File 4-12
Providing Help for Your Program 4-15
Creating P-Code Files 4-15

M-File Scripts and Functions 4-17
M-File Scripts 4-17
M-File Functions 4-18
Types of Functions 4-19
Identifying Dependencies 4-20

Function Handles 4-22
Constructing a Function Handle 4-22
Calling a Function Using Its Handle 4-23
Functions That Operate on Function Handles 4-25
Comparing Function Handles 4-25
Additional Information on Function Handles 4-30

Function Arguments 4-32
Overview 4-32
Checking the Number of Input Arguments 4-32
Passing Variable Numbers of Arguments 4-34
Parsing Inputs with inputParser 4-36
Passing Optional Arguments to Nested Functions 4-47
Returning Modified Input Arguments 4-50

Calling Functions 4-52

What Happens When You Call a Function 4-52
Determining Which Function Is Called 4-53
MATLAB Calling Syntax 4-56
Passing Certain Argument Types 4-60
Passing Arguments in Structures or Cell Arrays 4-62
Assigning Output Arguments 4-64
Calling External Functions 4-66
Running External Programs 4-67

5 Types of Functions

Overview of MATLAB Function Types **5-2**

Anonymous Functions **5-3**
Constructing an Anonymous Function 5-3
Arrays of Anonymous Functions 5-6
Outputs from Anonymous Functions 5-7
Variables Used in the Expression 5-8
Examples of Anonymous Functions 5-11

Primary M-File Functions **5-15**

Nested Functions **5-16**
Writing Nested Functions 5-16
Calling Nested Functions 5-17
Variable Scope in Nested Functions 5-19
Using Function Handles with Nested Functions 5-21
Restrictions on Assigning to Variables 5-26
Examples of Nested Functions 5-27

Subfunctions **5-33**
Overview 5-33
Calling Subfunctions 5-34
Accessing Help for a Subfunction 5-34

Private Functions **5-35**
Overview 5-35

Private Directories 5-35
Accessing Help for a Private Function 5-36

Overloaded Functions **5-37**

6 Data Import and Export

Overview **6-3**
File Types Supported by MATLAB 6-3
Other MATLAB I/O Capabilities 6-5
Functions Used in File Management 6-7

Supported File Formats **6-9**

Using the Import Wizard **6-11**
Overview 6-11
Starting the Import Wizard 6-11
Previewing Contents of the File or Clipboard [Text only] 6-13
Specifying Delimiters and Header Format [Text only] 6-14
Determining Assignment to Variables 6-15
Automated M-Code Generation 6-18
Writing Data to the Workspace 6-21

Accessing Files with Memory-Mapping **6-23**
Overview of Memory-Mapping in MATLAB 6-23
The memmapfile Class 6-27
Constructing a memmapfile Object 6-29
Reading a Mapped File 6-43
Writing to a Mapped File 6-48
Methods of the memmapfile Class 6-56
Deleting a Memory Map 6-58
Memory-Mapping Demo 6-58

Exporting Data to MAT-Files **6-64**
MAT-Files 6-64
Using the save Function 6-64
Saving Structures 6-65

Appending to an Existing File ... **6-66**
Data Compression ... **6-66**
Unicode Character Encoding ... **6-68**
Optional Output Formats ... **6-69**
Storage Requirements ... **6-70**
Saving From External Programs ... **6-71**

Importing Data From MAT-Files ... **6-72**
Using the load Function ... **6-72**
Previewing MAT-File Contents ... **6-72**
Loading Into a Structure ... **6-73**
Loading Binary Data ... **6-73**
Loading ASCII Data ... **6-74**

Importing Text Data ... **6-75**
The MATLAB Import Wizard ... **6-75**
Using Import Functions with Text Data ... **6-75**
Importing Numeric Text Data ... **6-78**
Importing Delimited ASCII Data Files ... **6-79**
Importing Numeric Data with Text Headers ... **6-80**
Importing Mixed Alphabetic and Numeric Data ... **6-81**
Importing from XML Documents ... **6-83**

Exporting Text Data ... **6-84**
Overview ... **6-84**
Exporting Delimited ASCII Data Files ... **6-86**
Using the diary Function to Export Data ... **6-87**
Exporting to XML Documents ... **6-88**

Working with Graphics Files ... **6-90**
Getting Information About Graphics Files ... **6-90**
Importing Graphics Data ... **6-91**
Exporting Graphics Data ... **6-91**

Working with Audio and Video Data ... **6-93**
Getting Information About Audio/Video Files ... **6-93**
Importing Audio/Video Data ... **6-94**
Exporting Audio/Video Data ... **6-95**

Working with Spreadsheets ... **6-98**
Microsoft Excel Spreadsheets ... **6-98**

Lotus 123 Spreadsheets ... **6-101**

Using Low-Level File I/O Functions ... **6-104**
Overview ... **6-104**
Opening Files ... **6-105**
Reading Binary Data ... **6-107**
Writing Binary Data ... **6-109**
Controlling Position in a File ... **6-109**
Reading Strings Line by Line from Text Files ... **6-112**
Reading Formatted ASCII Data ... **6-113**
Writing Formatted Text Files ... **6-114**
Closing a File ... **6-115**

Exchanging Files over the Internet ... **6-117**
Overview ... **6-117**
Downloading Web Content and Files ... **6-117**
Creating and Decompressing Zip Archives ... **6-119**
Sending E-Mail ... **6-120**
Performing FTP File Operations ... **6-122**

7 Working with Scientific Data Formats

Common Data Format (CDF) Files ... **7-2**
Getting Information About CDF Files ... **7-2**
Importing Data from a CDF File ... **7-3**
Exporting Data to a CDF File ... **7-6**

Flexible Image Transport System (FITS) Files ... **7-8**
Getting Information About FITS Files ... **7-8**
Importing Data from a FITS File ... **7-9**

Hierarchical Data Format (HDF5) Files ... **7-11**
Using the MATLAB High-Level HDF5 Functions ... **7-11**
Using the MATLAB Low-Level HDF5 Functions ... **7-26**

Hierarchical Data Format (HDF4) Files ... **7-36**
Using the HDF Import Tool ... **7-36**

Using the HDF Import Tool Subsetting Options 7-41
Using the MATLAB HDF4 High-Level Functions 7-53
Using the HDF4 Low-Level Functions 7-56

8 Error Handling

Error Reporting in MATLAB **8-2**
Overview **8-2**
Getting an Exception at the Command Line **8-2**
Getting an Exception in Your Program Code **8-3**
Generating a New Exception **8-4**

Capturing Information About the Error **8-5**
Overview **8-5**
The MException Class **8-5**
Properties of the MException Class **8-7**
Methods of the MException Class **8-14**

Throwing an Exception **8-16**

Responding to an Exception **8-17**
Overview **8-17**
The try-catch Statement **8-17**
Suggestions on How to Handle an Exception **8-19**

Warnings **8-22**
Reporting a Warning **8-22**
Identifying the Cause **8-23**

Warning Control **8-24**
Overview **8-24**
Warning Statements **8-25**
Warning Control Statements **8-26**
Output from Control Statements **8-28**
Saving and Restoring State **8-30**
Backtrace and Verbose Modes **8-31**

Debugging Errors and Warnings **8-34**

9 Classes and Objects

Classes and Objects: An Overview **9-2**
Overview **9-2**
Features of Object-Oriented Programming **9-3**
MATLAB Data Class Hierarchy **9-3**
Creating Objects **9-4**
Invoking Methods on Objects **9-4**
Private Methods **9-5**
Helper Functions **9-6**
Debugging Class Methods **9-6**
Setting Up Class Directories **9-6**
Data Structure **9-7**
Tips for C++ and Java Programmers **9-8**

Designing User Classes in MATLAB **9-9**
The MATLAB Canonical Class **9-9**
The Class Constructor Method **9-10**
Examples of Constructor Methods **9-12**
Identifying Objects Outside the Class Directory **9-12**
The display Method **9-13**
Accessing Object Data **9-13**
The set and get Methods **9-14**
Indexed Reference Using subsref and subsasgn **9-15**
Handling Subscripted Reference **9-16**
Handling Subscripted Assignment **9-19**
Object Indexing Within Methods **9-20**
Defining end Indexing for an Object **9-20**
Indexing an Object with Another Object **9-21**
Converter Methods **9-22**

Overloading Operators and Functions **9-23**
Overloading Operators **9-23**
Overloading Functions **9-25**

Example — A Polynomial Class **9-26**

Polynom Data Structure 9-26
Polynom Methods 9-26
The Polynom Constructor Method 9-27
Converter Methods for the Polynom Class 9-28
The Polynom display Method 9-30
The Polynom subsref Method 9-31
Overloading Arithmetic Operators for polynom 9-32
Overloading Functions for the Polynom Class 9-34
Listing Class Methods 9-36

Building on Other Classes 9-38
Overview 9-38
Simple Inheritance 9-38
Multiple Inheritance 9-40
Aggregation 9-40

Example — Assets and Asset Subclasses 9-41
Inheritance Model for the Asset Class 9-41
Asset Class Design 9-42
Other Asset Methods 9-43
The Asset Constructor Method 9-43
The Asset get Method 9-44
The Asset set Method 9-45
The Asset subsref Method 9-46
The Asset subsasgn Method 9-47
The Asset display Method 9-48
The Asset fieldcount Method 9-49
Designing the Stock Class 9-49
The Stock Constructor Method 9-50
The Stock get Method 9-52
The Stock set Method 9-53
The Stock subsref Method 9-54
The Stock subsasgn Method 9-55
The Stock display Method 9-57

Example — The Portfolio Container 9-58
Overview 9-58
Designing the Portfolio Class 9-58
The Portfolio Constructor Method 9-59
The Portfolio display Method 9-61
The Portfolio pie3 Method 9-61
Creating a Portfolio 9-62

Saving and Loading Objects **9-64**

Example — Defining saveobj and loadobj for Portfolio **9-65**
Methods Executed by Save and Load **9-65**
Summary of Code Changes **9-65**
The saveobj Method **9-66**
The loadobj Method **9-66**
Changing the Portfolio Constructor **9-67**
The Portfolio subsref Method **9-68**

Object Precedence **9-70**
How MATLAB Determines Precedence **9-70**
Specifying Precedence of User-Defined Classes **9-71**

How MATLAB Determines Which Method to Call **9-72**
Overview **9-72**
Selecting a Method **9-72**
Querying Which Method MATLAB Will Call **9-75**

10 Scheduling Program Execution with Timers

Using a MATLAB Timer Object **10-2**
Overview **10-2**
Example: Displaying a Message **10-3**

Creating Timer Objects **10-5**
Creating the Object **10-5**
Naming the Object **10-6**

Working with Timer Object Properties **10-7**
Retrieving the Value of Timer Object Properties **10-7**
Setting the Value of Timer Object Properties **10-8**

Starting and Stopping Timers **10-10**
Starting a Timer **10-10**
Starting a Timer at a Specified Time **10-10**

Stopping Timer Objects 10-11
Blocking the MATLAB Command Line 10-12

Creating and Executing Callback Functions **10-14**
Associating Commands with Timer Object Events 10-14
Creating Callback Functions 10-15
Specifying the Value of Callback Function Properties 10-17

Timer Object Execution Modes **10-19**
Executing a Timer Callback Function Once 10-19
Executing a Timer Callback Function Multiple Times 10-20
Handling Callback Function Queuing Conflicts 10-21

Deleting Timer Objects from Memory **10-23**
Deleting One or More Timer Objects 10-23
Testing the Validity of a Timer Object 10-23

Finding Timer Objects in Memory **10-24**
Finding All Timer Objects 10-24
Finding Invisible Timer Objects 10-24

11 Improving Performance and Memory Usage

Analyzing Your Program's Performance **11-2**
Overview 11-2
The M-File Profiler Utility 11-2
Stopwatch Timer Functions 11-2

Techniques for Improving Performance **11-4**
Vectorizing Loops 11-4
Preallocating Arrays 11-7
Use Distributed Arrays for Large Datasets 11-9
When Possible, Replace for with parfor (Parallel for) 11-9
Multithreading Capabilities in MATLAB 11-9
Limiting M-File Size and Complexity 11-9
Coding Loops in a MEX-File 11-10
Assigning to Variables 11-10

Operating on Real Data 11-11
Using Appropriate Logical Operators 11-11
Overloading Built-In Functions 11-12
Functions Are Generally Faster Than Scripts 11-12
Load and Save Are Faster Than File I/O Functions 11-12
Avoid Large Background Processes 11-12

Multiprocessing in MATLAB **11-13**
Overview 11-13
Implicit Multiprocessing 11-14
Explicit Multiprocessing 11-17

Memory Allocation in MATLAB **11-18**
Memory Allocation for Arrays 11-18
Data Structures and Memory 11-22

Memory Management Functions **11-24**

Strategies for Efficient Use of Memory **11-25**
Preallocating Arrays to Reduce Fragmentation 11-25
Allocating Large Matrices Earlier 11-26
Working with Large Amounts of Data 11-26

Resolving "Out of Memory" Errors **11-27**
General Suggestions for Reclaiming Memory 11-27
Compressing Data in Memory 11-28
Increasing System Swap Space 11-28
Freeing Up System Resources on Windows Systems 11-29
Reloading Variables on UNIX Systems 11-30

12 Programming Tips

Introduction **12-3**

Command and Function Syntax **12-4**
Syntax Help 12-4
Command and Function Syntaxes 12-4

Command Line Continuation 12-4
Completing Commands Using the Tab Key 12-5
Recalling Commands 12-5
Clearing Commands 12-6
Suppressing Output to the Screen 12-6

Help **12-7**
Using the Help Browser 12-7
Help on Functions from the Help Browser 12-8
Help on Functions from the Command Window 12-8
Topical Help 12-8
Paged Output 12-9
Writing Your Own Help 12-10
Help for Subfunctions and Private Functions 12-10
Help for Methods and Overloaded Functions 12-10

Development Environment **12-12**
Workspace Browser 12-12
Using the Find and Replace Utility 12-12
Commenting Out a Block of Code 12-13
Creating M-Files from Command History 12-13
Editing M-Files in EMACS 12-13

M-File Functions **12-14**
M-File Structure 12-14
Using Lowercase for Function Names 12-14
Getting a Function's Name and Path 12-15
What M-Files Does a Function Use? 12-15
Dependent Functions, Built-Ins, Classes 12-16

Function Arguments **12-17**
Getting the Input and Output Arguments 12-17
Variable Numbers of Arguments 12-17
String or Numeric Arguments 12-18
Passing Arguments in a Structure 12-18
Passing Arguments in a Cell Array 12-19

Program Development **12-20**
Planning the Program 12-20
Using Pseudo-Code 12-20
Selecting the Right Data Structures 12-20
General Coding Practices 12-21

Naming a Function Uniquely **12-21**
The Importance of Comments **12-21**
Coding in Steps **12-22**
Making Modifications in Steps **12-22**
Functions with One Calling Function **12-22**
Testing the Final Program **12-22**

Debugging **12-23**
The MATLAB Debug Functions **12-23**
More Debug Functions **12-23**
The MATLAB Graphical Debugger **12-24**
A Quick Way to Examine Variables **12-24**
Setting Breakpoints from the Command Line **12-25**
Finding Line Numbers to Set Breakpoints **12-25**
Stopping Execution on an Error or Warning **12-25**
Locating an Error from the Error Message **12-25**
Using Warnings to Help Debug **12-26**
Making Code Execution Visible **12-26**
Debugging Scripts **12-26**

Variables **12-27**
Rules for Variable Names **12-27**
Making Sure Variable Names Are Valid **12-27**
Do Not Use Function Names for Variables **12-28**
Checking for Reserved Keywords **12-28**
Avoid Using i and j for Variables **12-29**
Avoid Overwriting Variables in Scripts **12-29**
Persistent Variables **12-29**
Protecting Persistent Variables **12-29**
Global Variables **12-30**

Strings **12-31**
Creating Strings with Concatenation **12-31**
Comparing Methods of Concatenation **12-31**
Store Arrays of Strings in a Cell Array **12-32**
Converting Between Strings and Cell Arrays **12-32**
Search and Replace Using Regular Expressions **12-33**

Evaluating Expressions **12-34**
Find Alternatives to Using eval **12-34**
Assigning to a Series of Variables **12-34**
Short-Circuit Logical Operators **12-35**

Changing the Counter Variable within a for Loop **12-35**

MATLAB Path **12-36**
Precedence Rules **12-36**
File Precedence **12-37**
Adding a Directory to the Search Path **12-37**
Handles to Functions Not on the Path **12-37**
Making Toolbox File Changes Visible to MATLAB **12-38**
Making Nontoolbox File Changes Visible to MATLAB **12-39**
Change Notification on Windows **12-39**

Program Control **12-40**
Using break, continue, and return **12-40**
Using switch Versus if **12-41**
MATLAB case Evaluates Strings **12-41**
Multiple Conditions in a case Statement **12-41**
Implicit Break in switch-case **12-41**
Variable Scope in a switch **12-42**
Catching Errors with try-catch **12-42**
Nested try-catch Blocks **12-43**
Forcing an Early Return from a Function **12-43**

Save and Load **12-44**
Saving Data from the Workspace **12-44**
Loading Data into the Workspace **12-44**
Viewing Variables in a MAT-File **12-45**
Appending to a MAT-File **12-45**
Save and Load on Startup or Quit **12-46**
Saving to an ASCII File **12-46**

Files and Filenames **12-47**
Naming M-files **12-47**
Naming Other Files **12-47**
Passing Filenames as Arguments **12-48**
Passing Filenames to ASCII Files **12-48**
Determining Filenames at Run-Time **12-48**
Returning the Size of a File **12-48**

Input/Output **12-50**
File I/O Function Overview **12-50**
Common I/O Functions **12-50**
Readable File Formats **12-51**

Using the Import Wizard 12-51
Loading Mixed Format Data 12-51
Reading Files with Different Formats 12-52
Reading ASCII Data into a Cell Array 12-52
Interactive Input into Your Program 12-52

Starting MATLAB 12-53
Getting MATLAB to Start Up Faster 12-53

Operating System Compatibility 12-54
Executing O/S Commands from MATLAB 12-54
Searching Text with grep 12-54
Constructing Paths and Filenames 12-54
Finding the MATLAB Root Directory 12-55
Temporary Directories and Filenames 12-55

Demos 12-56
Demos Available with MATLAB 12-56

For More Information 12-57
Current CSSM 12-57
Archived CSSM 12-57
MATLAB Technical Support 12-57
Tech Notes 12-57
MATLAB Central 12-57
MATLAB Newsletters (Digest, News & Notes) 12-57
MATLAB Documentation 12-58
MATLAB Index of Examples 12-58

Index

Data Structures

Creating and Concatenating Matrices (p. 1-3)	Create a matrix or construct one from other matrices.
Matrix Indexing (p. 1-18)	Access or assign to elements of a matrix using methods of row and column indexing.
Getting Information About a Matrix (p. 1-28)	Retrieve information about the structure or contents of a matrix.
Resizing and Reshaping Matrices (p. 1-31)	Change the size, shape, or arrangement of elements in an existing matrix.
Shifting and Sorting Matrices (p. 1-41)	Shift matrix elements along one or more dimensions, or sort them into an ascending or descending order.
Operating on Diagonal Matrices (p. 1-46)	Construct and manipulate matrices along a diagonal of the rectangular shape.
Empty Matrices, Scalars, and Vectors (p. 1-48)	Work with matrices that have one or more dimensions equal to zero or one.
Full and Sparse Matrices (p. 1-54)	Conserve memory and get optimal performance with more efficient storage of matrices that contain a large number of zero values.

Multidimensional Arrays (p. 1-56)	Create and work with arrays that have more than two dimensions.
Summary of Matrix and Array Functions (p. 1-76)	Quick reference to the functions commonly used in working with matrices.

Creating and Concatenating Matrices

In this section...

"Overview" on page 1-3

"Constructing a Simple Matrix" on page 1-4

"Specialized Matrix Functions" on page 1-5

"Concatenating Matrices" on page 1-8

"Matrix Concatenation Functions" on page 1-9

"Generating a Numeric Sequence" on page 1-11

"Combining Unlike Data Types" on page 1-13

Overview

The most basic data structure in MATLAB® is the *matrix*: a two-dimensional, rectangularly shaped data structure capable of storing multiple elements of data in an easily accessible format. These data elements can be numbers, characters, logical states of `true` or `false`, or even other MATLAB structure types. MATLAB uses these two-dimensional matrices to store single numbers and linear series of numbers as well. In these cases, the dimensions are 1-by-1 and 1-by-n respectively, where `n` is the length of the numeric series. MATLAB also supports data structures that have more than two dimensions. These data structures are referred to as *arrays* in the MATLAB documentation.

MATLAB is a matrix-based computing environment. All of the data that you enter into MATLAB is stored in the form of a matrix or a multidimensional array. Even a single numeric value like `100` is stored as a matrix (in this case, a matrix having dimensions 1-by-1):

```
A = 100;

whos A
  Name      Size                    Bytes  Class

  A         1x1                         8  double array
```

Regardless of the data type being used, whether it is numeric, character, or logical `true` or `false` data, MATLAB stores this data in matrix (or array) form. For example, the string `'Hello World'` is a 1-by-11 matrix of individual character elements in MATLAB. You can also build matrices composed of more complex data types, such as MATLAB structures and cell arrays.

To create a matrix of basic data elements such as numbers or characters, see

- "Constructing a Simple Matrix" on page 1-4
- "Specialized Matrix Functions" on page 1-5

To build a matrix composed of other matrices, see

- "Concatenating Matrices" on page 1-8
- "Matrix Concatenation Functions" on page 1-9

This section also describes

- "Generating a Numeric Sequence" on page 1-11
- "Combining Unlike Data Types" on page 1-13

Constructing a Simple Matrix

The simplest way to create a matrix in MATLAB is to use the matrix constructor operator, `[]`. Create a row in the matrix by entering elements (shown as `E` below) within the brackets. Separate each element with a comma or space:

```
row = [E1, E2, ..., Em]          row = [E1 E2 ... Em]
```

For example, to create a one row matrix of five elements, type

```
A = [12 62 93 -8 22];
```

To start a new row, terminate the current row with a semicolon:

```
A = [row1; row2; ...; rown]
```

This example constructs a 3 row, 5 column (or 3-by-5) matrix of numbers. Note that all rows must have the same number of elements:

```
A = [12 62 93 -8 22; 16 2 87 43 91; -4 17 -72 95 6]
A =
    12    62    93    -8    22
    16     2    87    43    91
    -4    17   -72    95     6
```

The square brackets operator constructs two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To construct arrays of more than two dimensions, see “Creating Multidimensional Arrays” on page 1-58.

For instructions on how to read or overwrite any matrix element, see “Matrix Indexing” on page 1-18.

Entering Signed Numbers

When entering signed numbers into a matrix, make sure that the sign immediately precedes the numeric value. Note that while the following two expressions are equivalent,

```
7 -2 +5                                7 - 2 + 5
ans =                                    ans =
    10                                       10
```

the next two are *not*:

```
[7 -2 +5]                              [7 - 2 + 5]
ans =                                    ans =
     7    -2     5                           10
```

Specialized Matrix Functions

MATLAB has a number of functions that create different kinds of matrices. Some create specialized matrices like the Hankel or Vandermonde matrix. The functions shown in the table below create matrices for more general use.

Function	Description
ones	Create a matrix or array of all ones.
zeros	Create a matrix or array of all zeros.
eye	Create a matrix with ones on the diagonal and zeros elsewhere.
accumarray	Distribute elements of an input matrix to specified locations in an output matrix, also allowing for accumulation.
diag	Create a diagonal matrix from a vector.
magic	Create a square matrix with rows, columns, and diagonals that add up to the same number.
rand	Create a matrix or array of uniformly distributed random numbers.
randn	Create a matrix or array of normally distributed random numbers and arrays.
randperm	Create a vector (1-by-n matrix) containing a random permutation of the specified integers.

Most of these functions return matrices of type `double` (double-precision floating point). However, you can easily build basic arrays of any numeric type using the `ones`, `zeros`, and `eye` functions.

To do this, specify the MATLAB class name as the last argument:

```
A = zeros(4, 6, 'uint32')
A =
          0          0          0          0          0          0
          0          0          0          0          0          0
          0          0          0          0          0          0
          0          0          0          0          0          0
```

Examples

Here are some examples of how you can use these functions.

Creating a Magic Square Matrix. A magic square is a matrix in which the sum of the elements in each column, or each row, or each main diagonal is the same. To create a 5-by-5 magic square matrix, use the `magic` function as shown.

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Note that the elements of each row, each column, and each main diagonal add up to the same value: 65.

Creating a Random Matrix. The `rand` function creates a matrix or array with elements uniformly distributed between zero and one. This example multiplies each element by 20:

```
A = rand(5) * 20
A =
   19.0026   15.2419   12.3086    8.1141    1.1578
    4.6228    9.1294   15.8387   18.7094    7.0574
   12.1369    0.3701   18.4363   18.3381   16.2633
    9.7196   16.4281   14.7641    8.2054    0.1972
   17.8260    8.8941    3.5253   17.8730    2.7778
```

The sequence of numbers produced by `rand` is determined by the internal state of the generator. Setting the generator to the same fixed state enables you to repeat computations. Examples in this documentation that use the `rand` function are initialized to a state of 0 to make the output consistent each time they are run:

```
rand('state', 0);
```

Creating a Diagonal Matrix. Use `diag` to create a diagonal matrix from a vector. You can place the vector along the main diagonal of the matrix, or on a diagonal that is above or below the main one, as shown here. The `-1` input places the vector one row below the main diagonal:

```
A = [12 62 93 -8 22];

B = diag(A, -1)
B =
     0     0     0     0     0     0
    12     0     0     0     0     0
     0    62     0     0     0     0
     0     0    93     0     0     0
     0     0     0    -8     0     0
     0     0     0     0    22     0
```

Concatenating Matrices

Matrix concatenation is the process of joining one or more matrices to make a new matrix. The brackets `[]` operator discussed earlier in this section serves not only as a matrix constructor, but also as the MATLAB concatenation operator. The expression `C = [A B]` horizontally concatenates matrices `A` and `B`. The expression `C = [A; B]` vertically concatenates them.

This example constructs a new matrix `C` by concatenating matrices `A` and `B` in a vertical direction:

```
A = ones(2, 5) * 6;        % 2-by-5 matrix of 6's
B = rand(3, 5);            % 3-by-5 matrix of random values

C = [A; B]                 % Vertically concatenate A and B
C =
    6.0000    6.0000    6.0000    6.0000    6.0000
    6.0000    6.0000    6.0000    6.0000    6.0000
    0.9501    0.4860    0.4565    0.4447    0.9218
    0.2311    0.8913    0.0185    0.6154    0.7382
    0.6068    0.7621    0.8214    0.7919    0.1763
```

Keeping Matrices Rectangular

You can construct matrices, or even multidimensional arrays, using concatenation as long as the resulting matrix does not have an irregular shape (as in the second illustration shown below). If you are building a matrix horizontally, then each component matrix must have the same number of

rows. When building vertically, each component must have the same number of columns.

This diagram shows two matrices of the same height (i.e., same number of rows) being combined horizontally to form a new matrix.

$$\underset{\text{3-by-2}}{\begin{bmatrix} 7 & 23 \\ 41 & 11 \\ -1 & 90 \end{bmatrix}} + \underset{\text{3-by-4}}{\begin{bmatrix} 46 & 0 & 13 & -4 \\ 44 & 62 & 31 & 98 \\ 3 & 51 & -9 & 25 \end{bmatrix}} = \underset{\text{3-by-6}}{\begin{bmatrix} 7 & 23 & 46 & 0 & 13 & -4 \\ 41 & 11 & 44 & 62 & 31 & 98 \\ -1 & 90 & 3 & 51 & -9 & 25 \end{bmatrix}}$$

The next diagram illustrates an attempt to horizontally combine two matrices of unequal height. MATLAB does not allow this.

$$\underset{\text{3-by-2}}{\begin{bmatrix} 7 & 23 \\ 41 & 11 \\ -1 & 90 \end{bmatrix}} + \underset{\text{2-by-4}}{\begin{bmatrix} 46 & 0 & 13 & -4 \\ 44 & 62 & 31 & 98 \end{bmatrix}} \neq \begin{bmatrix} 7 & 23 & 46 & 0 & 13 & -4 \\ 41 & 11 & 44 & 62 & 31 & 98 \\ -1 & 90 & & & & \end{bmatrix}$$

Matrix Concatenation Functions

The following functions combine existing matrices to form a new matrix.

Function	Description
`cat`	Concatenate matrices along the specified dimension
`horzcat`	Horizontally concatenate matrices
`vertcat`	Vertically concatenate matrices
`repmat`	Create a new matrix by replicating and tiling existing matrices
`blkdiag`	Create a block diagonal matrix from existing matrices

Examples

Here are some examples of how you can use these functions.

Concatenating Matrices and Arrays. An alternative to using the `[]` operator for concatenation are the three functions `cat`, `horzcat`, and `vertcat`. With these functions, you can construct matrices (or multidimensional arrays) along a specified dimension. Either of the following commands accomplish the same task as the command `C = [A; B]` used in the section on "Concatenating Matrices" on page 1-8:

```
C = cat(1, A, B);       % Concatenate along the first dimension
C = vertcat(A, B);      % Concatenate vertically
```

Replicating a Matrix. Use the `repmat` function to create a matrix composed of copies of an existing matrix. When you enter

```
repmat(M, v, h)
```

MATLAB replicates input matrix `M` `v` times vertically and `h` times horizontally. For example, to replicate existing matrix `A` into a new matrix `B`, use

```
A = [8 1 6; 3 5 7; 4 9 2]
A =
   8   1   6
   3   5   7
   4   9   2

B = repmat(A, 2, 4)
B =
   8   1   6   8   1   6   8   1   6   8   1   6
   3   5   7   3   5   7   3   5   7   3   5   7
   4   9   2   4   9   2   4   9   2   4   9   2
   8   1   6   8   1   6   8   1   6   8   1   6
   3   5   7   3   5   7   3   5   7   3   5   7
   4   9   2   4   9   2   4   9   2   4   9   2
```

Creating a Block Diagonal Matrix. The `blkdiag` function combines matrices in a diagonal direction, creating what is called a block diagonal matrix. All other elements of the newly created matrix are set to zero:

```
A = magic(3);
B = [-5 -6 -9; -4 -4 -2];
C = eye(2) * 8;
```

```
D = blkdiag(A, B, C)
D =
    8    1    6    0    0    0    0    0
    3    5    7    0    0    0    0    0
    4    9    2    0    0    0    0    0
    0    0    0   -5   -6   -9    0    0
    0    0    0   -4   -4   -2    0    0
    0    0    0    0    0    0    8    0
    0    0    0    0    0    0    0    8
```

Generating a Numeric Sequence

Because numeric sequences can often be useful in constructing and indexing into matrices and arrays, MATLAB provides a special operator to assist in creating them.

This section covers

- "The Colon Operator" on page 1-11
- "Using the Colon Operator with a Step Value" on page 1-12

The Colon Operator

The colon operator (`first:last`) generates a 1-by-n matrix (or *vector*) of sequential numbers from the `first` value to the `last`. The default sequence is made up of incremental values, each 1 greater than the previous one:

```
A = 10:15
A =
     10     11     12     13     14     15
```

The numeric sequence does not have to be made up of positive integers. It can include negative numbers and fractional numbers as well:

```
A = -2.5:2.5
A =
   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

By default, MATLAB always increments by exactly 1 when creating the sequence, even if the ending value is not an integral distance from the start:

```
A = 1:6.3
A =
     1     2     3     4     5     6
```

Also, the default series generated by the colon operator always increments rather than decrementing. The operation shown in this example attempts to increment from 9 to 1 and thus MATLAB returns an empty matrix:

```
A = 9:1
A =
   Empty matrix: 1-by-0
```

The next section explains how to generate a nondefault numeric series.

Using the Colon Operator with a Step Value

To generate a series that does not use the default of incrementing by 1, specify an additional value with the colon operator (`first:step:last`). In between the starting and ending value is a `step` value that tells MATLAB how much to increment (or decrement, if `step` is negative) between each number it generates.

To generate a series of numbers from 10 to 50, incrementing by 5, use

```
A = 10:5:50
A =
    10    15    20    25    30    35    40    45    50
```

You can increment by noninteger values. This example increments by `0.2`:

```
A = 3:0.2:3.8
A =
    3.0000    3.2000    3.4000    3.6000    3.8000
```

To create a sequence with a decrementing interval, specify a negative step value:

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

Combining Unlike Data Types

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike data types when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See Chapter 2, "Data Types" for information on any of the MATLAB data types discussed here.)

Data type conversion is done with respect to a preset precedence of data types. The following table shows the five data types you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a `double` and `single` matrix always yields a matrix of type `single`. MATLAB converts the `double` element to `single` to accomplish this.

Combining Unlike Integer Types

If you combine different integer types in a matrix (e.g., signed with unsigned, or 8-bit integers with 16-bit integers), MATLAB returns a matrix in which all elements are of one common type. MATLAB sets all elements of the resulting matrix to the data type of the left-most element in the input matrix. For example, the result of the following concatenation is a vector of three 16-bit signed integers:

```
A = [int16(450) uint8(250) int32(1000000)]
```

MATLAB also displays a warning to inform you that the result may not be what you had expected:

```
A = [int16(450) uint8(250) int32(1000000)];
Warning: Concatenation with dominant (left-most) integer class
may overflow other operands on conversion to return class.
```

You can disable this warning by entering the following two commands directly after the operation that caused the warning. The first command retrieves the message identifier associated with the most recent warning issued by MATLAB. The second command uses this identifier to disable any further warnings of that type from being issued:

```
[msg, intcat_msgid] = lastwarn;
warning('off', intcat_msgid);
```

To reenable the warning so that it will now be displayed, use

```
warning('on', intcat_msgid);
```

You can use these commands to disable or enable the display of any MATLAB warning.

Example of Combining Unlike Integer Sizes. After disabling the integer concatenation warnings as shown above, concatenate the following two numbers once, and then switch their order. The return value depends on the order in which the integers are concatenated. The left-most type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
A =
   5000   50

B = [int8(50) int16(5000)]
B =
   50   127
```

The first operation returns a vector of 16-bit integers. The second returns a vector of 8-bit integers. The element `int16(5000)` is set to `127`, the maximum value for an 8-bit signed integer.

The same rules apply to vertical concatenation:

```
C = [int8(50); int16(5000)]
```

```
C =
     50
    127
```

Note You can find the maximum or minimum values for any MATLAB integer type using the `intmax` and `intmin` functions. For floating-point types, use `realmax` and `realmin`.

Example of Combining Signed with Unsigned. Now do the same exercise with signed and unsigned integers. Again, the left-most element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
A =
   -100   100

B = [uint8(100) int8(-100)]
B =
   100   0
```

The element `int8(-100)` is set to zero because it is no longer signed.

MATLAB evaluates each element *prior to* concatenating them into a combined array. In other words, the following statement evaluates to an 8-bit signed integer (equal to 50) and an 8-bit unsigned integer (unsigned -50 is set to zero) before the two elements are combined. Following the concatenation, the second element retains its zero value but takes on the unsigned `int8` type:

```
A = [int8(50), uint8(-50)]
A =
  50    0
```

Combining Integer and Noninteger Data

If you combine integers with `double`, `single`, or `logical` data types, all elements of the resulting matrix are given the data type of the left-most integer. For example, all elements of the following vector are set to `int32`:

```
A = [true pi int32(1000000) single(17.32) uint8(250)]
```

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Concatenation Examples

Here are some examples of data type conversion during matrix construction.

Combining Single and Double Types. Combining `single` values with `double` values yields a `single` matrix. Note that `5.73*10^300` is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value):

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000   -2.8000    3.1416       Inf

class(x)              % Display the data type of x
ans =
   single
```

Combining Integer and Double Types. Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer):

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21   -22    23     3     7
```

```
class(x)
ans =
   int8
```

Combining Character and Double Types. Combining `character` values with `double` values yields a `character` matrix. MATLAB converts the `double` elements in this example to their `character` equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
   ABCDEF

class(x)
ans =
   char
```

Combining Logical and Double Types. Combining `logical` values with `double` values yields a `double` matrix. MATLAB converts the `logical` `true` and `false` elements in this example to `double`:

```
x = [true false false pi sqrt(7)]
x =
    1.0000         0         0    3.1416    2.6458

class(x)
ans =
   double
```

Matrix Indexing

In this section...

"Accessing Single Elements" on page 1-18

"Linear Indexing" on page 1-19

"Functions That Control Indexing Style" on page 1-19

"Accessing Multiple Elements" on page 1-20

"Using Logicals in Array Indexing" on page 1-22

"Single-Colon Indexing with Different Array Types" on page 1-26

"Indexing on Assignment" on page 1-26

Accessing Single Elements

To reference a particular element in a matrix, specify its row and column number using the following syntax, where `A` is the matrix variable. Always specify the row first and column second:

```
A(row, column)
```

For example, for a 4-by-4 magic square `A`,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

you would access the element at row 4, column 2 with

```
A(4, 2)
ans =
    14
```

For arrays with more than two dimensions, specify additional indices following the `row` and `column` indices. See the section on "Multidimensional Arrays" on page 1-56.

Linear Indexing

With MATLAB, you can refer to the elements of a matrix with a single subscript, `A(k)`. MATLAB stores matrices and arrays not in the shape that they appear when displayed in the MATLAB Command Window, but as a single column of elements. This single column is composed of all of the columns from the matrix, each appended to the last.

So, matrix `A`

```
A = [2 6 9; 4 2 8; 3 5 1]
A =
     2     6     9
     4     2     8
     3     5     1
```

is actually stored in memory as the sequence

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

The element at row 3, column 2 of matrix `A` (value = `5`) can also be identified as element 6 in the actual storage sequence. To access this element, you have a choice of using the standard `A(3,2)` syntax, or you can use `A(6)`, which is referred to as *linear indexing*.

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

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

Given the expression `A(3,2)`, MATLAB calculates the offset into `A`'s storage column as `(2-1) * 3 + 3`, or `6`. Counting down six elements in the column accesses the value `5`.

Functions That Control Indexing Style

If you have row-column subscripts but want to use linear indexing instead, you can convert to the latter using the `sub2ind` function. In the 3-by-3 matrix

A used in the previous section, sub2ind changes a standard row-column index of (3,2) to a linear index of 6:

```
A = [2 6 9; 4 2 8; 3 5 1];

linearindex = sub2ind(size(A), 3, 2)
linearindex =
     6
```

To get the row-column equivalent of a linear index, use the ind2sub function:

```
[row col] = ind2sub(size(A), 6)
row =
     3
col =
     2
```

Accessing Multiple Elements

For the 4-by-4 matrix A shown below, it is possible to compute the sum of the elements in the fourth column of A by typing

```
A = magic(4);
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

You can reduce the size of this expression using the colon operator. Subscript expressions involving colons refer to portions of a matrix. The expression

```
A(1:m, n)
```

refers to the elements in rows 1 through m of column n of matrix A. Using this notation, you can compute the sum of the fourth column of A more succinctly:

```
sum(A(1:4, 4))
```

Nonconsecutive Elements

To refer to nonconsecutive elements in a matrix, use the colon operator with a step value. The m:3:n in this expression means to make the assignment to every third element in the matrix. Note that this example uses linear indexing:

```
B = A;

B(1:3:16) = -10
B =
   -10     2     3   -10
     5    11   -10     8
     9   -10     6    12
   -10    14    15   -10
```

MATLAB supports a type of array indexing that uses one array as the index into another array. You can base this type of indexing on either the values or the positions of elements in the indexing array.

Here is an example of value-based indexing where array `B` indexes into elements `1`, `3`, `6`, `7`, and `10` of array `A`. In this case, the *numeric values* of array `B` designate the intended elements of `A`:

```
A = 5:5:50
A =
     5    10    15    20    25    30    35    40    45    50
B = [1 3 6 7 10];

A(B)
ans =
     5    15    30    35    50
```

The end Keyword

MATLAB provides the keyword `end` to designate the last element in a particular dimension of an array. This keyword can be useful in instances where your program does not know how many rows or columns there are in a matrix. You can replace the expression in the previous example with

```
B(1:3:end) = -10
```

Note The keyword `end` has several meanings in MATLAB. It can be used as explained above, or to terminate a conditional block of code such as `if` and `for` blocks, or to terminate a nested function.

Specifying All Elements of a Row or Column

The colon by itself refers to *all* the elements in a row or column of a matrix. Using the following syntax, you can compute the sum of all elements in the second column of a 4-by-4 magic square A:

```
sum(A(:, 2))
ans =
    34
```

By using the colon with linear indexing, you can refer to all elements in the entire matrix. This example displays all the elements of matrix A, returning them in a column-wise order:

```
A(:)
ans =
    16
     5
     9
     4
     .
     .
     .
    12
     1
```

Using Logicals in Array Indexing

A logical array index designates the elements of an array A based on their *position* in the indexing array, B, not their value. In this *masking* type of operation, every true element in the indexing array is treated as a positional index into the array being accessed.

In the following example, B is a matrix of logical ones and zeros. The position of these elements in B determines which elements of A are designated by the expression A(B):

```
A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
```

```
B = logical([0 1 0; 1 0 1; 0 0 1]);
B =
     0     1     0
     1     0     1
     0     0     1

A(B)
ans =
     4
     2
     6
     9
```

The `find` function can be useful with logical arrays as it returns the linear indices of nonzero elements in `B`, and thus helps to interpret `A(B)`:

```
find(B)
ans =
     2
     4
     8
     9
```

Logical Indexing – Example 1

This example creates logical array `B` that satisfies the condition `A > 0.5`, and uses the positions of ones in `B` to index into `A`:

```
rand('twister', 5489);     % Initialize the state of the
                           % random number generator.
A = rand(5);
B = A > 0.5;

A(B) = 0
A =
         0    0.0975    0.1576    0.1419         0
         0    0.2785         0    0.4218    0.0357
    0.1270         0         0         0         0
         0         0    0.4854         0         0
         0         0         0         0         0
```

A simpler way to express this is

```
A(A > 0.5) = 0
```

Logical Indexing – Example 2

The next example highlights the location of the prime numbers in a magic square using logical indexing to set the nonprimes to 0:

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

B = isprime(A)
B =
     0     1     1     1
     1     1     0     0
     0     1     0     0
     0     0     0     0

A(~B) = 0;                          % Logical indexing

A
A =
     0     2     3    13
     5    11     0     0
     0     7     0     0
     0     0     0     0

find(B)
ans =
     2
     5
     6
     7
     9
    13
```

Logical Indexing with a Smaller Array

In most cases, the logical indexing array should have the same number of elements as the array being indexed into, but this is not a requirement. The indexing array may have smaller (but not larger) dimensions:

```
A = [1 2 3;4 5 6;7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

B = logical([0 1 0; 1 0 1])
B =
     0     1     0
     1     0     1

isequal(numel(A), numel(B))
ans =
     0

A(B)
ans =
     4
     7
     8
```

MATLAB treats the missing elements of the indexing array as if they were present and set to zero, as in array `C` below:

```
% Add zeros to indexing array C to give it the same number of
% elements as A.
C = logical([B(:);0;0;0]);

isequal(numel(A), numel(C))
ans =
     1

A(C)
ans =
     4
```

```
7
8
```

Single-Colon Indexing with Different Array Types

When you index into a standard MATLAB array using a single colon, MATLAB returns a column vector (see variable `n`, below). When you index into a structure or cell array using a single colon, you get a comma-separated list “Comma-Separated Lists” on page 3-79 (see variables `c` and `s`, below).

Create three types of arrays:

```
n = [1 2 3; 4 5 6];
c = {1 2; 3 4};
s = cell2struct(c, {'a', 'b'}, 1);  s(:,2)=s(:,1);
```

Use single-colon indexing on each:

```
n(:)               c{:}               s(:).a
ans =              ans =              ans =
     1                  1                  1
     4             ans =              ans =
     2                  3                  2
     5             ans =              ans =
     3                  2                  1
     6             ans =              ans =
                        4                  2
```

Indexing on Assignment

When assigning values from one matrix to another matrix, you can use any of the styles of indexing covered in this section. Matrix assignment statements also have the following requirement.

In the assignment `A(J,K,...)   = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, etc. may be scalar, vector, or array, provided that all of the following are true:

- The number of subscripts specified for `B`, not including trailing subscripts equal to 1, does not exceed `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5, 1:4, 1, 2)`

`= B(5:8)` is valid because both sides of the equation use one nonscalar subscript.

- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4, 3, 3:9) = B(5:8, 1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

Getting Information About a Matrix

In this section...
"Dimensions of the Matrix" on page 1-28
"Data Types Used in the Matrix" on page 1-29
"Data Structures Used in the Matrix" on page 1-30

Dimensions of the Matrix

These functions return information about the shape and size of a matrix.

Function	Description
`length`	Return the length of the longest dimension. (The length of a matrix or array with any zero dimension is zero.)
`ndims`	Return the number of dimensions.
`numel`	Return the number of elements.
`size`	Return the length of each dimension.

The following examples show some simple ways to use these functions. Both use the 3-by-5 matrix `A` shown here:

```
rand('state', 0);     % Initialize random number generator
A = rand(5) * 10;
A(4:5, :) = []
A =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
```

Example Using numel

Using the `numel` function, find the average of all values in matrix `A`:

```
sum(A(:))/numel(A)
ans =
```

```
    5.8909
```

Example Using ndims, numel, and size

Using `ndims` and `size`, go through the matrix and find those values that are between 5 and 7, inclusive:

```
if ndims(A) ~= 2
   return
end

[rows cols] = size(A);
for m = 1:rows
   for n = 1:cols
      x = A(m, n);
      if x >= 5 && x <= 7
         disp(sprintf('A(%d, %d) = %5.2f', m, n, A(m,n)))
      end
   end
end
```

The code returns the following:

```
A(1, 3) =  6.15
A(3, 1) =  6.07
```

Data Types Used in the Matrix

These functions test elements of a matrix for a specific data type.

Function	Description
`isa`	Detect if input is of a given data type.
`iscell`	Determine if input is a cell array.
`iscellstr`	Determine if input is a cell array of strings.
`ischar`	Determine if input is a character array.
`isfloat`	Determine if input is a floating-point array.
`isinteger`	Determine if input is an integer array.

Function	Description
`islogical`	Determine if input is a logical array.
`isnumeric`	Determine if input is a numeric array.
`isreal`	Determine if input is an array of real numbers.
`isstruct`	Determine if input is a MATLAB structure array.

Example Using isnumeric and isreal

Pick out the real numeric elements from this vector:

```
A = [5+7i 8/7 4.23 39j pi 9-2i];

for m = 1:numel(A)
   if isnumeric(A(m)) && isreal(A(m))
      disp(A(m))
   end
end
```

The values returned are

```
    1.1429
    4.2300
    3.1416
```

Data Structures Used in the Matrix

These functions test elements of a matrix for a specific data structure.

Function	Description
`isempty`	Determine if input has any dimension with size zero.
`isscalar`	Determine if input is a 1-by-1 matrix.
`issparse`	Determine if input is a sparse matrix.
`isvector`	Determine if input is a 1-by-n or n-by-1 matrix.

Resizing and Reshaping Matrices

In this section...
"Expanding the Size of a Matrix" on page 1-31
"Diminishing the Size of a Matrix" on page 1-35
"Reshaping a Matrix" on page 1-36
"Preallocating Memory" on page 1-38

Expanding the Size of a Matrix

You can expand the size of any existing matrix as long as doing so does not give the resulting matrix an irregular shape. (See "Keeping Matrices Rectangular" on page 1-8). For example, you can vertically combine a 4-by-3 matrix and 7-by-3 matrix because all rows of the resulting matrix have the same number of columns (3).

Two ways of expanding the size of an existing matrix are

- Concatenating new elements onto the matrix
- Storing to a location outside the bounds of the matrix

Note If you intend to expand the size of a matrix repeatedly over time as it requires more room (usually done in a programming loop), it is advisable to preallocate space for the matrix when you initially create it. See "Preallocating Memory" on page 1-38.

Concatenating Onto the Matrix

Concatenation is most useful when you want to expand a matrix by adding new elements or blocks that are compatible in size with the original matrix. This means that the size of all matrices being joined along a specific dimension must be equal along that dimension. See "Concatenating Matrices" on page 1-8.

This example runs a user-defined function compareResults on the data in matrices stats04 and stats03. Each time through the loop, it concatenates the results of this function onto the end of the data stored in comp04:

```
col = 10;
comp04 = [];

for k = 1:50
   t = compareResults(stats04(k,1:col), stats03(k,1:col));
   comp04 = [comp04; t];
end
```

Concatenating to a Structure or Cell Array. You can add on to arrays of structures or cells in the same way as you do with ordinary matrices. This example creates a 3-by-8 matrix of structures S, each having 3 fields: x, y, and z, and then concatenates a second structure matrix S2 onto the original:

Create a 3-by-8 structure array S:

```
for k = 1:24
   S(k) = struct('x', 10*k, 'y', 10*k+1, 'z', 10*k+2);
end
S = reshape(S, 3, 8);
```

Create a second array that is 3-by-2 and uses the same field names:

```
for k = 25:30
   S2(k-24) = struct('x', 10*k, 'y', 10*k+1, 'z', 10*k+2);
end
S2= reshape(S2, 3, 2);
```

Concatenate S2 onto S along the horizontal dimension:

```
S = [S S2]
S =
3x10 struct array with fields:
    x
    y
    z
```

Adding Smaller Blocks to a Matrix

To add one or more elements to a matrix where the sizes are not compatible, you can often just store the new elements outside the boundaries of the original matrix. MATLAB automatically pads the matrix with zeros to keep it rectangular.

Construct a 3-by-5 matrix, and attempt to add a new element to it using concatenation. The operation fails because you are attempting to join a one-column matrix with one that has five columns:

```
A = [ 10  20  30  40  50; ...
      60  70  80  90 100; ...
     110 120 130 140 150];

A = [A; 160]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

Try this again, but this time do it in such a way that enables MATLAB to make adjustments to the size of the matrix. Store the new element in row 4, a row that does not yet exist in this matrix. MATLAB expands matrix `A` by an entire new row by padding columns 2 through 5 with zeros:

```
A(4,1) = 160
A =
    10     20     30     40     50
    60     70     80     90    100
   110    120    130    140    150
   160      0      0      0      0
```

Note Attempting to read from nonexistent matrix locations generates an error. You can only write to these locations.

You can also expand the matrix by adding a matrix instead of just a single element:

```
A(4:6,1:3) = magic(3)+100
```

```
A =
    10    20    30    40    50
    60    70    80    90   100
   110   120   130   140   150
   108   101   106     0     0
   103   105   107     0     0
   104   109   102     0     0
```

You do not have to add new elements sequentially. Wherever you store the new elements, MATLAB pads with zeros to make the resulting matrix rectangular in shape:

```
A(4,8) = 300
A =
    10    20    30    40    50     0     0     0
    60    70    80    90   100     0     0     0
   110   120   130   140   150     0     0     0
     0     0     0     0     0     0     0   300
```

Expanding a Structure or Cell Array. You can expand a structure or cell array in the same way that you can a matrix. This example adds an additional cell to a cell array by storing it beyond the bounds of the original array. MATLAB pads the data structure with empty cells (`[]`) to keep it rectangular.

The original array is 2-by-3:

```
C = {'Madison', 'G', [5 28 1967]; ...
     46, '325 Maple Dr', 3015.28}
```

Add a cell to `C{3,1}` and MATLAB appends an entire row:

```
C{3, 1} = ...
struct('Fund_A', .45, 'Fund_E', .35, 'Fund_G', 20);
C =
    'Madison'        'G'                 [1x3 double]
    [        46]    '325 Maple Dr'    [3.0153e+003]
    [1x1 struct]                  []               []
```

Expanding a Character Array. You can expand character arrays in the same manner as other MATLAB arrays, but it is generally not recommended. MATLAB expands any array by padding uninitialized elements with zeros. Because zero is interpreted by MATLAB and some other programming languages as a string terminator, you may find that some functions treat the expanded string as if it were less than its full length.

Expand a 1-by-5 character array to twelve characters. The result appears at first to be a typical string:

```
greeting = 'Hello';    greeting(1,8:12) = 'World'
greeting =
   Hello  World
```

Closer inspection however reveals string terminators at the point of expansion:

```
uint8(greeting)
ans =
   72   101   108   108   111     0     0    87   111   114   108   100
```

This causes some functions, like `strcmp`, to return what might be considered an unexpected result:

```
strcmp(greeting, 'Hello  World')
ans =
     0
```

Diminishing the Size of a Matrix

You can delete rows and columns from a matrix by assigning the empty array `[]` to those rows or columns. Start with

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Then, delete the second column of `A` using

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

This changes matrix A to

```
A =
    16     3    13
     5    10     8
     9     6    12
     4    15     1
```

If you delete a single element from a matrix, the result is not a matrix anymore. So expressions like

```
A(1,2) = []
```

result in an error. However, you can use linear indexing to delete a single element, or a sequence of elements. This reshapes the remaining elements into a row vector:

```
A(2:2:10) = []
```

results in

```
A =
    16     9     3     6    13    12     1
```

Reshaping a Matrix

The following functions change the shape of a matrix.

Function	Description
`reshape`	Modify the shape of a matrix.
`rot90`	Rotate the matrix by 90 degrees.
`fliplr`	Flip the matrix about a vertical axis.
`flipud`	Flip the matrix about a horizontal axis.
`flipdim`	Flip the matrix along the specified direction.

Function	Description
transpose	Flip a matrix about its main diagonal, turning row vectors into column vectors and vice versa.
ctranspose	Transpose a matrix and replace each element with its complex conjugate.

Examples

Here are a few examples to illustrate some of the ways you can reshape matrices.

Reshaping a Matrix. Reshape 3-by-4 matrix `A` to have dimensions 2-by-6:

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

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

Transposing a Matrix. Transpose `A` so that the row elements become columns. You can use either the `transpose` function or the transpose operator (`.'`) to do this:

```
B = A.'
B =
      1      2      3
      4      5      6
      7      8      9
     10     11     12
```

There is a separate function called ctransposethat performs a complex conjugate transpose of a matrix. The equivalent operator for `ctranpose` on a matrix `A` is `A'`:

```
A = [1+9i 2-8i 3+7i; 4-6i 5+5i 6-4i]
A =
   1.0000 + 9.0000i   2.0000 -8.0000i   3.0000 + 7.0000i
   4.0000 -6.0000i   5.0000 + 5.0000i   6.0000 -4.0000i

B = A'
B =
   1.0000 -9.0000i   4.0000 + 6.0000i
   2.0000 + 8.0000i   5.0000 -5.0000i
   3.0000 -7.0000i   6.0000 + 4.0000i
```

Rotating a Matrix. Rotate the matrix by 90 degrees:

```
B = rot90(A)
B =
    10    11    12
     7     8     9
     4     5     6
     1     2     3
```

Flipping a Matrix. Flip `A` in a left-to-right direction:

```
B = fliplr(A)
B =
    10     7     4     1
    11     8     5     2
    12     9     6     3
```

Preallocating Memory

Repeatedly expanding the size of an array over time, (for example, adding more elements to it each time through a programming loop), can adversely affect the performance of your program. This is because

- MATLAB has to spend time allocating more memory each time you increase the size of the array.
- This newly allocated memory is likely to be noncontiguous, thus slowing down any operations that MATLAB needs to perform on the array.

The preferred method for sizing an array that is expected to grow over time is to estimate the maximum possible size for the array, and preallocate this amount of memory for it at the time the array is created. In this way, your program performs one memory allocation that reserves one contiguous block.

The following command preallocates enough space for a 25,000 by 10,000 matrix, and initializes each element to zero:

```
A = zeros(25000, 10000);
```

Building a Preallocated Array

Once memory has been preallocated for the maximum estimated size of the array, you can store your data in the array as you need it, each time appending to the existing data. This example preallocates a large array, and then reads blocks of data from a file into the array until it gets to the end of the file:

```
blocksize = 5000;
maxrows = 2500000; cols = 20;
rp = 1;     % row pointer

% Preallocate A to its maximum possible size
A = zeros(maxrows, cols);

% Open the data file, saving the file pointer.
fid = fopen('statfile.dat', 'r');

while true
   % Read from file into a cell array.  Stop at EOF.
   block = textscan(fid, '%n', blocksize*cols);
   if isempty(block{1})   break,   end;

   % Convert cell array to matrix, reshape, place into A.
   A(rp:rp+blocksize-1, 1:cols) = ...
      reshape(cell2mat(block), blocksize, cols);

   % Process the data in A.
   evaluate_stats(A);              % User-defined function

   % Update row pointer
```

```
    rp = rp + blocksize;
end
```

Note If you eventually need more room in a matrix than you had preallocated, you can preallocate additional storage in the same manner, and concatenate this additional storage onto the original array.

Shifting and Sorting Matrices

In this section...
"Shift and Sort Functions" on page 1-41
"Shifting the Location of Matrix Elements" on page 1-41
"Sorting the Data in Each Column" on page 1-43
"Sorting the Data in Each Row" on page 1-43
"Sorting Row Vectors" on page 1-44

Shift and Sort Functions

Use these functions to shift or sort the elements of a matrix.

Function	Description
`circshift`	Circularly shift matrix contents.
`sort`	Sort array elements in ascending or descending order.
`sortrows`	Sort rows in ascending order.
`issorted`	Determine if matrix elements are in sorted order.

You can sort matrices, multidimensional arrays, and cell arrays of strings along any dimension and in ascending or descending order of the elements. The sort functions also return an optional array of indices showing the order in which elements were rearranged during the sorting operation.

Shifting the Location of Matrix Elements

The `circshift` function shifts the elements of a matrix in a circular manner along one or more dimensions. Rows or columns that are shifted out of the matrix circulate back into the opposite end. For example, shifting a 4-by-7 matrix one place to the left moves the elements in columns 2 through 7 to columns 1 through 6, and moves column 1 to column 7.

Create a 5-by-8 matrix named `A` and shift it to the right along the second (horizontal) dimension by three places. (You would use `[0, -3]` to shift to the left by three places):

```
A = [1:8; 11:18; 21:28; 31:38; 41:48]
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48

B = circshift(A, [0, 3])
B =
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
```

Now take A and shift it along both dimensions: three columns to the right and two rows up:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48];

B = circshift(A, [-2, 3])
B =
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
```

Since circshift circulates shifted rows and columns around to the other end of a matrix, shifting by the exact size of A returns all rows and columns to their original location:

```
B = circshift(A, size(A));

all(B(:) == A(:))           % Do all elements of B equal A?
ans =
     1                      % Yes
```

Sorting the Data in Each Column

The `sort` function sorts matrix elements along a specified dimension. The syntax for the function is

```
sort(matrix, dimension)
```

To sort the columns of a matrix, specify 1 as the `dimension` argument. To sort along rows, specify `dimension` as 2.

This example first constructs a 6-by-7 random matrix:

```
rand('state', 0);     % Initialize random number generator
A = floor(rand(6,7) * 100);
A =
    95    45    92    41    13     1    84
    23     1    73    89    20    74    52
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    89    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

Sort each column of `A` in ascending order:

```
c = sort(A, 1)
c =
    23     1    17     0    13     1     1
    48    44    40     5    19    41    20
    60    45    73    35    19    44    52
    76    61    91    41    20    46    67
    89    79    92    81    27    74    83
    95    82    93    89    60    93    84

issorted(c(:, 1))
ans =
     1
```

Sorting the Data in Each Row

Use `issorted` to sort data in each row. Using the example above, if you sort each row of `A` in descending order, `issorted` tests for an ascending sequence. You can flip the vector to test for a sorted descending sequence:

```
rand('state', 0);     A = floor(rand(6,7) * 100);

r = sort(A, 2, 'descend')
r =
    95    92    84    45    41    13     1
    89    74    73    52    23    20     1
    82    60    44    20    19    17     5
    93    67    60    48    44    40    35
    93    89    83    81    61    46    27
    91    79    76    41    19     1     0

issorted(fliplr(r(1, :)))
ans =
     1
```

When you specify a second output, `sort` returns the indices of the original matrix `A` positioned in the order they appear in the output matrix. In this next example, the second row of `index` contains the sequence `4 3 2 5 1`, which means that the sorted elements in output matrix `r` were taken from `A(2,4)`, `A(2,3)`, `A(2,2)`, `A(2,5)`, and `A(2,1)`:

```
[r index] = sort(A, 2, 'descend');
index
index =
     1     3     7     2     4     5     6
     4     6     3     7     1     5     2
     2     1     6     7     5     3     4
     6     7     5     1     2     3     4
     3     1     7     4     2     6     5
     3     2     1     6     5     7     4
```

Sorting Row Vectors

The `sortrows` function keeps the elements of each row in its original order, but sorts the entire row of vectors according to the order of the elements in the specified column.

The next example creates a random matrix `A`:

```
rand('state', 0);      % Initialize random number generator
A = floor(rand(6,7) * 100);
A =
    95    45    92    41    13     1    84
    23     1    73    89    20    74    52
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    89    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

To sort in ascending order based on the values in column 1, you can call `sortrows` with just the one input argument:

```
sortrows(A)
r =
    23     1    73    89    20    74    52
    48    44    40    35    60    93    67
    60    82    17     5    19    44    20
    76    79    91     0    19    41     1
    89    61    93    81    27    46    83
    95    45    92    41    13     1    84
```

To base the sort on a column other than the first, call `sortrows` with a second input argument that indicates the column number, column 4 in this case:

```
r = sortrows(A, 4)
r =
    76    79    91     0    19    41     1
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    95    45    92    41    13     1    84
    89    61    93    81    27    46    83
    23     1    73    89    20    74    52
```

Operating on Diagonal Matrices

In this section...
"Diagonal Matrix Functions" on page 1-46
"Constructing a Matrix from a Diagonal Vector" on page 1-46
"Returning a Triangular Portion of a Matrix" on page 1-47
"Concatenating Matrices Diagonally" on page 1-47

Diagonal Matrix Functions

There are several MATLAB functions that work specifically on diagonal matrices.

Function	Description
`blkdiag`	Construct a block diagonal matrix from input arguments.
`diag`	Return a diagonal matrix or the diagonals of a matrix.
`trace`	Compute the sum of the elements on the main diagonal.
`tril`	Return the lower triangular part of a matrix.
`triu`	Return the upper triangular part of a matrix.

Constructing a Matrix from a Diagonal Vector

The `diag` function has two operations that it can perform. You can use it to generate a diagonal matrix:

```
A = diag([12:4:32])
A =
    12     0     0     0     0     0
     0    16     0     0     0     0
     0     0    20     0     0     0
     0     0     0    24     0     0
     0     0     0     0    28     0
     0     0     0     0     0    32
```

You can also use the diag function to scan an existing matrix and return the values found along one of the diagonals:

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

diag(A, 2)        % Return contents of second diagonal of A
ans =
     1
    14
    22
```

Returning a Triangular Portion of a Matrix

The tril and triu functions return a triangular portion of a matrix, the former returning the piece from the lower left and the latter from the upper right. By default, the main diagonal of the matrix divides these two segments. You can use an alternate diagonal by specifying an offset from the main diagonal as a second input argument:

```
A = magic(6);

B = tril(A, -1)
B =
     0     0     0     0     0     0
     3     0     0     0     0     0
    31     9     0     0     0     0
     8    28    33     0     0     0
    30     5    34    12     0     0
     4    36    29    13    18     0
```

Concatenating Matrices Diagonally

You can diagonally concatenate matrices to form a composite matrix using the blkdiag function. See “Creating a Block Diagonal Matrix” on page 1-10 for more information on how this works.

Empty Matrices, Scalars, and Vectors

In this section...

"Overview" on page 1-48

"The Empty Matrix" on page 1-49

"Scalars" on page 1-51

"Vectors" on page 1-52

Overview

Although matrices are two dimensional, they do not always appear to have a rectangular shape. A 1-by-8 matrix, for example, has two dimensions yet is linear. These matrices are described in the following sections:

- "The Empty Matrix" on page 1-49

  An *empty matrix* has one of more dimensions that are equal to zero. A two-dimensional matrix with both dimensions equal to zero appears in MATLAB as `[]`. The expression `A = []` assigns a 0-by-0 empty matrix to `A`.

- "Scalars" on page 1-51

  A *scalar* is 1-by-1 and appears in MATLAB as a single real or complex number (e.g., `7`, `583.62`, `-3.51`, `5.46097e-14`, `83+4i`).

- "Vectors" on page 1-52

  A *vector* is 1-by-n or n-by-1, and appears in MATLAB as a row or column of real or complex numbers:

```
Column Vector                Row Vector

     53.2                 53.2 87.39 4-12i 43.9
     87.39
     4-12i
     43.9
```

The Empty Matrix

A matrix having at least one dimension equal to zero is called an empty matrix. The simplest empty matrix is 0-by-0 in size. Examples of more complex matrices are those of dimension 0-by-5 or 10-by-0.

To create a 0-by-0 matrix, use the square bracket operators with no value specified:

```
A = [];

whos A
  Name      Size         Bytes  Class

  A         0x0              0  double array
```

You can create empty matrices (and arrays) of other sizes using the `zeros`, `ones`, `rand`, or `eye` functions. To create a 0-by-5 matrix, for example, use

```
A = zeros(0,5)
```

Operating on an Empty Matrix

The basic model for empty matrices is that any operation that is defined for `m`-by-`n` matrices, and that produces a result whose dimension is some function of `m` and `n`, should still be allowed when `m` or `n` is zero. The size of the result of this operation is consistent with the size of the result generated when working with nonempty values, but instead is evaluated at zero.

For example, horizontal concatenation

```
C = [A B]
```

requires that `A` and `B` have the same number of rows. So if `A` is `m`-by-`n` and `B` is `m`-by-`p`, then `C` is `m-by-(n+p)`. This is still true if `m` or `n` or `p` is zero.

As with all matrices in MATLAB, you must follow the rules concerning compatible dimensions. In the following example, an attempt to add a 1-by-3 matrix to a 0-by-3 empty matrix results in an error:

```
[1 2 3] + ones(0,3)
??? Error using ==> +
Matrix dimensions must agree.
```

Common Operations. The following operations on an empty scalar array return zero:

```
A = [];
size(A), length(A), numel(A), any(A), sum(A)
```

These operations on an empty scalar array return a nonzero value:

```
A = [];
ndims(A), isnumeric(A), isreal(A), isfloat(A), isempty(A), ...
   all(A), prod(A)
```

Using Empty Matrices in Relational Operations

You can use empty matrices in relational operations such as "equal to" (`==`) or "greater than" (`>`) as long as both operands have the same dimensions, or the nonempty operand is scalar. The result of any relational operation involving an empty matrix is the empty matrix. Even comparing an empty matrix for equality to itself does not return `true`, but instead yields an empty matrix:

```
x = ones(0,3);
y = x;

y == x
ans =
   Empty matrix: 0-by-3
```

Using Empty Matrices in Logical Operations

MATLAB has two distinct types of logical operators:

- Short-circuit (`&&`, `||`) — Used in testing multiple logical conditions (e.g., `x >= 50 && x < 100`) where each condition evaluates to a scalar `true` or `false`.
- Element-wise (`&`, `|`) — Performs a logical AND, OR, or NOT on each element of a matrix or array.

Short-circuit Operations. The rule for operands used in short-circuit operations is that each operand must be convertible to a logical scalar value. Because of this rule, empty matrices cannot be used in short-circuit logical operations. Such operations return an error.

The only exception is in the case where MATLAB can determine the result of a logical statement without having to evaluate the entire expression. This is true for the following two statements because the result of the entire statements are known by considering just the first term:

```
true || []
ans =
   1

false && []
ans =
   0
```

Elementwise Operations. Unlike the short-circuit operators, all elementwise operations on empty matrices are considered valid as long as the dimensions of the operands agree, or the nonempty operand is scalar. Element-wise operations on empty matrices always return an empty matrix:

```
true | []
ans =
   []
```

Note This behavior is consistent with the way MATLAB does scalar expansion with binary operators, wherein the nonscalar operand determines the size of the result.

•

Scalars

Any individual real or complex number is represented in MATLAB as a 1-by-1 matrix called a scalar value:

```
A = 5;

ndims(A)          % Check number of dimensions in A
ans =
     2

size(A)           % Check value of row and column dimensions
ans =
     1     1
```

Use the `isscalar` function to tell if a variable holds a scalar value:

```
isscalar(A)
ans =
    1
```

Vectors

Matrices with one dimension equal to one and the other greater than one are called vectors. Here is an example of a numeric vector:

```
A = [5.73 2-4i 9/7 25e3 .046 sqrt(32) 8j];

size(A)           % Check value of row and column dimensions
ans =
     1     7
```

You can construct a vector out of other vectors, as long as the critical dimensions agree. All components of a row vector must be scalars or other row vectors. Similarly, all components of a column vector must be scalars or other column vectors:

```
A = [29 43 77 9 21];
B = [0 46 11];

C = [A 5 ones(1,3) B]
C =
   29   43   77    9   21    5    1    1    1    0   46   11
```

Concatenating an empty matrix to a vector has no effect on the resulting vector. The empty matrix is ignored in this case:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Use the `isvector` function to tell if a variable holds a vector:

```
isvector(A)
ans =
    1
```

Full and Sparse Matrices

In this section...
“Overview” on page 1-54
“Sparse Matrix Functions” on page 1-54

Overview

It is not uncommon to have matrices with a large number of zero-valued elements and, because MATLAB stores zeros in the same way it stores any other numeric value, these elements can use memory space unnecessarily and can sometimes require extra computing time.

Sparse matrices provide a way to store data that has a large percentage of zero elements more efficiently. While *full matrices* internally store every element in memory regardless of value, *sparse matrices* store only the nonzero elements and their row indices. Using sparse matrices can significantly reduce the amount of memory required for data storage.

You can create sparse matrices for the `double` and `logical` data types. All MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

See the section on Sparse Matrices in the MATLAB Mathematics documentation for more information on working with sparse matrices.

Sparse Matrix Functions

This table shows some of the functions most commonly used when working with sparse matrices.

Function	Description
`full`	Convert a sparse matrix to a full matrix.
`issparse`	Determine if a matrix is sparse.

Function	Description
`nnz`	Return the number of nonzero matrix elements.
`nonzeros`	Return the nonzero elements of a matrix.
`nzmax`	Return the amount of storage allocated for nonzero elements.
`spalloc`	Allocate space for a sparse matrix.
`sparse`	Create a sparse matrix or convert full to sparse.
`speye`	Create a sparse identity matrix.
`sprand`	Create a sparse uniformly distributed random matrix.

Multidimensional Arrays

In this section...

"Overview" on page 1-56
"Creating Multidimensional Arrays" on page 1-58
"Accessing Multidimensional Array Properties" on page 1-62
"Indexing Multidimensional Arrays" on page 1-62
"Reshaping Multidimensional Arrays" on page 1-66
"Permuting Array Dimensions" on page 1-68
"Computing with Multidimensional Arrays" on page 1-70
"Organizing Data in Multidimensional Arrays" on page 1-71
"Multidimensional Cell Arrays" on page 1-73
"Multidimensional Structure Arrays" on page 1-74

Overview

An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.

column

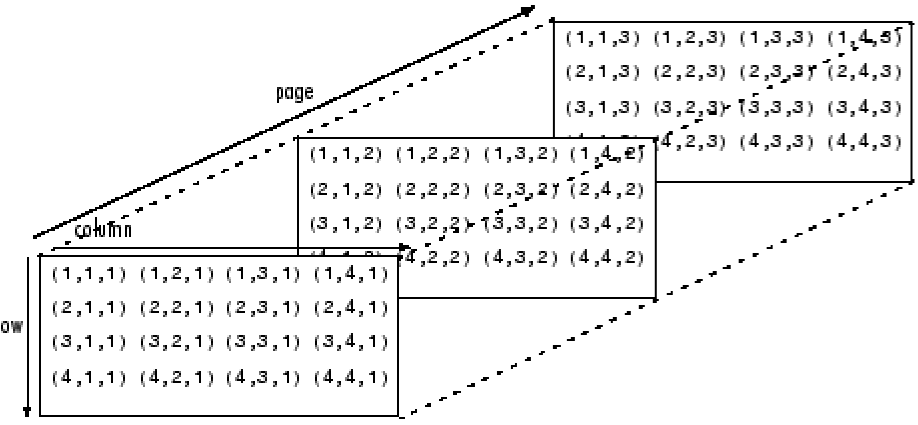
row

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

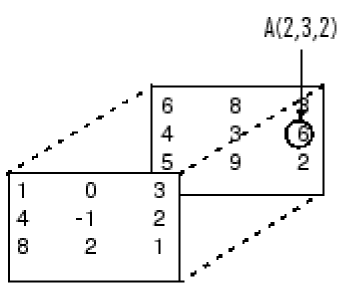
You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index.

Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This illustration uses the concept of a *page* to represent dimensions 3 and higher.



To access the element in the second row, third column of page 2, for example, you use the subscripts `(2,3,2)`.




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

     1     0     3
     4    -1     2
     8     2     1


A(:,:,2) =

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

As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two reference a row-column pair; the second two access the third and fourth dimensions of data.

Most of the operations that you can perform on matrices (i.e., two-dimensional arrays) can also be done on multidimensional arrays.

Note The general multidimensional array functions reside in the `datatypes` directory.

Creating Multidimensional Arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. In addition, MATLAB provides a special concatenation function that is useful for building multidimensional arrays.

This section discusses

- "Generating Arrays Using Indexing" on page 1-58
- "Extending Multidimensional Arrays" on page 1-59
- "Generating Arrays Using MATLAB Functions" on page 1-60
- "Building Multidimensional Arrays with the cat Function" on page 1-60

Generating Arrays Using Indexing

One way to create a multidimensional array is to create a two-dimensional array and extend it. For example, begin with a simple two-dimensional array `A`.

```
A = [5 7 8; 0 1 9; 4 3 6];
```

`A` is a 3-by-3 array, that is, its row dimension is 3 and its column dimension is 3. To add a third dimension to `A`,

```
A(:,:,2) = [1 0 4; 3 5 6; 9 8 7]
```

MATLAB responds with

```
A(:,:,1) =
     5     7     8
     0     1     9
     4     3     6

A(:,:,2) =
     1     0     4
     3     5     6
     9     8     7
```

You can continue to add rows, columns, or pages to the array using similar assignment statements.

Extending Multidimensional Arrays

To extend `A` in any dimension:

- Increment or add the appropriate subscript and assign the desired values.
- Assign the same number of elements to corresponding array dimensions. For numeric arrays, all rows must have the same number of elements, all pages must have the same number of rows and columns, and so on.

You can take advantage of the MATLAB scalar expansion capabilities, together with the colon operator, to fill an entire dimension with a single value:

```
A(:,:,3) = 5;

A(:,:,3)
ans =
     5     5     5
     5     5     5
     5     5     5
```

To turn `A` into a 3-by-3-by-3-by-2, four-dimensional array, enter

```
A(:,:,1,2) = [1 2 3; 4 5 6; 7 8 9];
A(:,:,2,2) = [9 8 7; 6 5 4; 3 2 1];
A(:,:,3,2) = [1 0 1; 1 1 0; 0 1 1];
```

Note that after the first two assignments MATLAB pads A with zeros, as needed, to maintain the corresponding sizes of dimensions.

Generating Arrays Using MATLAB Functions

You can use MATLAB functions such as randn, ones, and zeros to generate multidimensional arrays in the same way you use them for two-dimensional arrays. Each argument you supply represents the size of the corresponding dimension in the resulting array. For example, to create a 4-by-3-by-2 array of normally distributed random numbers:

```
B = randn(4,3,2)
```

To generate an array filled with a single constant value, use the repmat function. repmat replicates an array (in this case, a 1-by-1 array) through a vector of array dimensions.

```
B = repmat(5, [3 4 2])

B(:,:,1) =
     5     5     5     5
     5     5     5     5
     5     5     5     5

B(:,:,2) =
     5     5     5     5
     5     5     5     5
     5     5     5     5
```

Note Any dimension of an array can have size zero, making it a form of empty array. For example, 10-by-0-by-20 is a valid size for a multidimensional array.

Building Multidimensional Arrays with the cat Function

The cat function is a simple way to build multidimensional arrays; it concatenates a list of arrays along a specified dimension:

```
B = cat(dim, A1, A2...)
```

where A1, A2, and so on are the arrays to concatenate, and dim is the dimension along which to concatenate the arrays.

For example, to create a new array with cat:

```
B = cat(3, [2 8; 0 5], [1 3; 7 9])

B(:,:,1) =
     2     8
     0     5

B(:,:,2) =
     1     3
     7     9
```

The cat function accepts any combination of existing and new data. In addition, you can nest calls to cat. The lines below, for example, create a four-dimensional array.

```
A = cat(3, [9 2; 6 5], [7 1; 8 4])
B = cat(3, [3 5; 0 1], [5 6; 2 1])
D = cat(4, A, B, cat(3, [1 2; 3 4], [4 3;2 1]))
```

cat automatically adds subscripts of 1 between dimensions, if necessary. For example, to create a 2-by-2-by-1-by-2 array, enter

```
C = cat(4, [1 2; 4 5], [7 8; 3 2])
```

In the previous case, cat inserts as many singleton dimensions as needed to create a four-dimensional array whose last dimension is not a singleton dimension. If the dim argument had been 5, the previous statement would have produced a 2-by-2-by-1-by-1-by-2 array. This adds additional 1s to indexing expressions for the array. To access the value 8 in the four-dimensional case, use

Accessing Multidimensional Array Properties

You can use the following MATLAB functions to get information about multidimensional arrays you have created.

- `size` — Returns the size of each array dimension.

```
size(C)
ans =
     2       2       1       2
   rows  columns   dim3    dim4
```

- `ndims` — Returns the number of dimensions in the array.

```
ndims(C)
ans =
     4
```

- `whos` — Provides information on the format and storage of the array.

```
whos
Name      Size        Bytes   Class

A         2x2x2          64   double array
B         2x2x2          64   double array
C         4-D            64   double array
D         4-D           192   double array

Grand total is 48 elements using 384 bytes
```

Indexing Multidimensional Arrays

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well.

To access a single element of a multidimensional array, use integer subscripts. Each subscript indexes a dimension—the first indexes the row dimension, the second indexes the column dimension, the third indexes the first page dimension, and so on.

Consider a 10-by-5-by-3 array `nddata` of random integers:

```
nddata = fix(8 * randn(10,5,3));
```

To access element (3,2) on page 2 of nddata, for example, use nddata(3,2,2).

You can use vectors as array subscripts. In this case, each vector element must be a valid subscript, that is, within the bounds defined by the dimensions of the array. To access elements (2,1), (2,3), and (2,4) on page 3 of nddata, use

```
nddata(2,[1 3 4],3);
```

The Colon and Multidimensional Array Indexing

The MATLAB colon indexing extends to multidimensional arrays. For example, to access the entire third column on page 2 of nddata, use nddata(:,3,2).

The colon operator is also useful for accessing other subsets of data. For example, nddata(2:3,2:3,1) results in a 2-by-2 array, a subset of the data on page 1 of nddata. This matrix consists of the data in rows 2 and 3, columns 2 and 3, on the first page of the array.

The colon operator can appear as an array subscript on both sides of an assignment statement. For example, to create a 4-by-4 array of zeros:

```
C = zeros(4, 4)
```

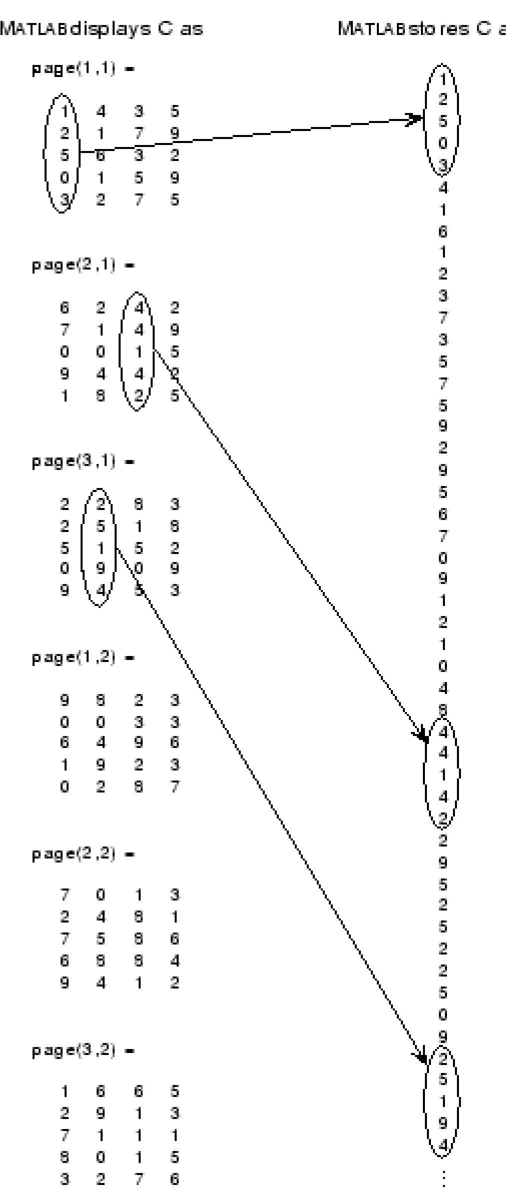
Now assign a 2-by-2 subset of array nddata to the four elements in the center of C.

```
C(2:3,2:3) = nddata(2:3,1:2,2)
```

Linear Indexing with Multidimensional Arrays

MATLAB linear indexing also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise. See “Linear Indexing” on page 1-19 for an introduction to this topic.

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



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

```
ans =
     0
```

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

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

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

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

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

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

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

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

```
C(3,2,1,1,1,1,1,1)
```

is equivalent to

```
C(3,2)
```

Avoiding Ambiguity in Multidimensional Indexing

Some assignment statements, such as

```
A(:,:,2) = 1:10
```

are ambiguous because they do not provide enough information about the shape of the dimension to receive the data. In the case above, the statement tries to assign a one-dimensional vector to a two-dimensional destination. MATLAB produces an error for such cases. To resolve the ambiguity, be sure you provide enough information about the destination for the assigned data, and that both data and destination have the same shape. For example:

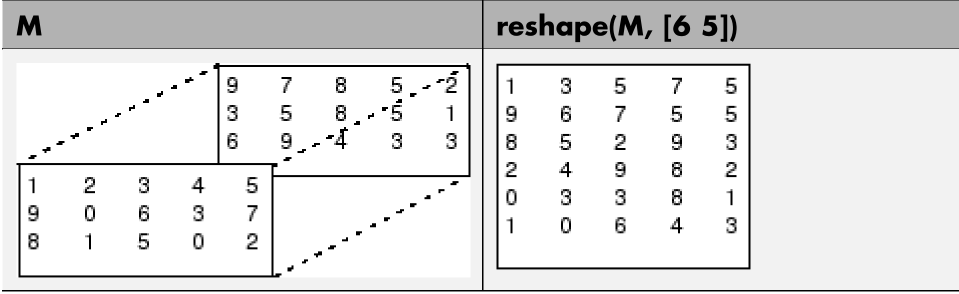
```
A(1,:,2) = 1:10;
```

Reshaping Multidimensional Arrays

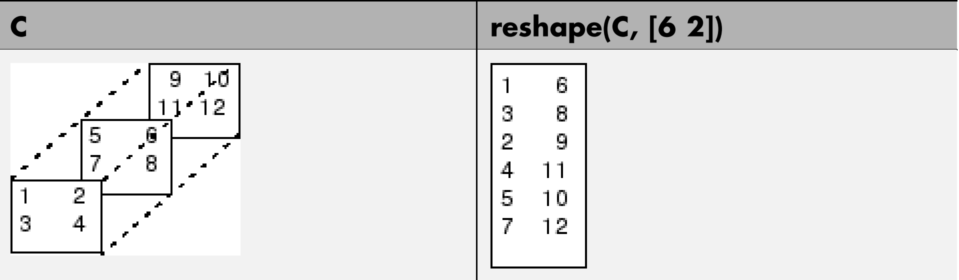
Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by respecifying the array's row, column, or page dimensions while retaining the same elements. The `reshape` function performs the latter operation. For multidimensional arrays, its form is

```
B = reshape(A,[s1 s2 s3 ...])
```

`s1`, `s2`, and so on represent the desired size for each dimension of the reshaped matrix. Note that a reshaped array must have the same number of elements as the original array (that is, the product of the dimension sizes is constant).



The `reshape` function operates in a columnwise manner. It creates the reshaped matrix by taking consecutive elements down each column of the original data construct.



Here are several new arrays from reshaping `nddata`:

```
B = reshape(nddata, [6 25])
C = reshape(nddata, [5 3 10])
D = reshape(nddata, [5 3 2 5])
```

Removing Singleton Dimensions

MATLAB creates singleton dimensions if you explicitly specify them when you create or reshape an array, or if you perform a calculation that results in an array dimension of one:

```
B = repmat(5, [2 3 1 4]);

size(B)
ans =
     2     3     1     4
```

The `squeeze` function removes singleton dimensions from an array:

```
C = squeeze(B);

size(C)
ans =
     2     3     4
```

The `squeeze` function does not affect two-dimensional arrays; row vectors remain rows.

Permuting Array Dimensions

The `permute` function reorders the dimensions of an array:

```
B = permute(A, dims);
```

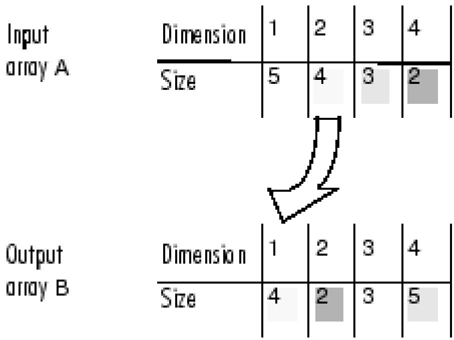
`dims` is a vector specifying the new order for the dimensions of `A`, where 1 corresponds to the first dimension (rows), 2 corresponds to the second dimension (columns), 3 corresponds to pages, and so on.

A	B = permute(A, [2 1 3])	C = permute(A, [3 2 1])
A(:,:,1) = 1 2 3 4 5 6 7 8 9 A(:,:,2) = 0 5 4 2 7 6 9 3 1	B(:,:,1) = 1 4 7 2 5 8 3 6 9 B(:,:,2) = 0 2 9 5 7 3 4 6 1 Row and column subscripts are reversed (page-by-page transposition).	C(:,:,1) = 1 2 3 0 5 4 C(:,:,2) = 4 5 6 2 7 6 C(:,:,3) = 7 8 9 9 3 1 Row and page subscripts are reversed.

For a more detailed look at the `permute` function, consider a four-dimensional array `A` of size 5-by-4-by-3-by-2. Rearrange the dimensions, placing the column dimension first, followed by the second page dimension, the first page dimension, then the row dimension. The result is a 4-by-2-by-3-by-5 array.

```
B = permute(A,[2 4 3 1])
```

Move dimension 2 of A to first subscript position of B, dimension 4 to second subscript position, and so on.



The order of dimensions in permute's argument list determines the size and shape of the output array. In this example, the second dimension moves to the first position. Because the second dimension of the original array had size 4, the output array's first dimension also has size 4.

You can think of `permute`'s operation as an extension of the `transpose` function, which switches the row and column dimensions of a matrix. For `permute`, the order of the input dimension list determines the reordering of the subscripts. In the example above, element `(4,2,1,2)` of `A` becomes element `(2,2,1,4)` of `B`, element `(5,4,3,2)` of `A` becomes element `(4,2,3,5)` of `B`, and so on.

Inverse Permutation

The `ipermute` function is the inverse of `permute`. Given an input array `A` and a vector of dimensions `v`, `ipermute` produces an array `B` such that `permute(B,v)` returns `A`.

For example, these statements create an array `E` that is equal to the input array `C`:

```
D = ipermute(C, [1 4 2 3]);
E = permute(D, [1 4 2 3])
```

You can obtain the original array after permuting it by calling `ipermute` with the same vector of dimensions.

Computing with Multidimensional Arrays

Many of the MATLAB computational and mathematical functions accept multidimensional arrays as arguments. These functions operate on specific dimensions of multidimensional arrays; that is, they operate on individual elements, on vectors, or on matrices.

Operating on Vectors

Functions that operate on vectors, like `sum`, `mean`, and so on, by default typically work on the first nonsingleton dimension of a multidimensional array. Most of these functions optionally let you specify a particular dimension on which to operate. There are exceptions, however. For example, the `cross` function, which finds the cross product of two vectors, works on the first nonsingleton dimension having length 3.

Note In many cases, these functions have other restrictions on the input arguments — for example, some functions that accept multiple arrays require that the arrays be the same size. Refer to the online help for details on function arguments.

Operating Element-by-Element

MATLAB functions that operate element-by-element on two-dimensional arrays, like the trigonometric and exponential functions in the `elfun` directory, work in exactly the same way for multidimensional cases. For example, the `sin` function returns an array the same size as the function's input argument. Each element of the output array is the sine of the corresponding element of the input array.

Similarly, the arithmetic, logical, and relational operators all work with corresponding elements of multidimensional arrays that are the same size in every dimension. If one operand is a scalar and one an array, the operator applies the scalar to each element of the array.

Operating on Planes and Matrices

Functions that operate on planes or matrices, such as the linear algebra and matrix functions in the `matfun` directory, do not accept multidimensional

arrays as arguments. That is, you cannot use the functions in the matfun directory, or the array operators *, ^, \, or /, with multidimensional arguments. Supplying multidimensional arguments or operands in these cases results in an error.

You can use indexing to apply a matrix function or operator to matrices within a multidimensional array. For example, create a three-dimensional array A:

```
A = cat(3, [1 2 3; 9 8 7; 4 6 5], [0 3 2; 8 8 4; 5 3 5], ...
    [6 4 7; 6 8 5; 5 4 3]);
```

Applying the eig function to the entire multidimensional array results in an error:

```
eig(A)
??? Error using ==> eig
Input arguments must be 2-D.
```

You can, however, apply eig to planes within the array. For example, use colon notation to index just one page (in this case, the second) of the array:

```
eig(A(:,:,2))
ans =
   12.9129
   -2.6260
    2.7131
```

Note In the first case, subscripts are not colons; you must use squeeze to avoid an error. For example, eig(A(2,:,:)) results in an error because the size of the input is [1 3 3]. The expression eig(squeeze(A(2,:,:))), however, passes a valid two-dimensional matrix to eig.

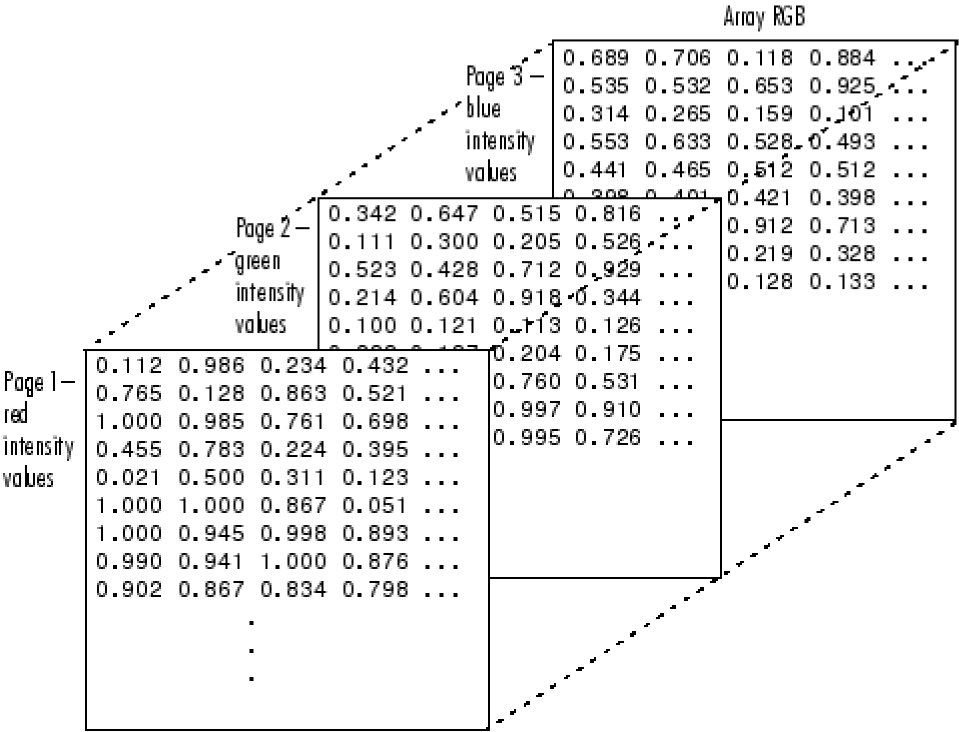
Organizing Data in Multidimensional Arrays

You can use multidimensional arrays to represent data in two ways:

- As planes or pages of two-dimensional data. You can then treat these pages as matrices.

- As multivariate or multidimensional data. For example, you might have a four-dimensional array where each element corresponds to either a temperature or air pressure measurement taken at one of a set of equally spaced points in a room.

For example, consider an RGB image. For a single image, a multidimensional array is probably the easiest way to store and access data.



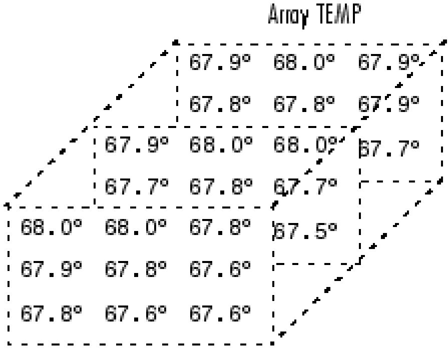
To access an entire plane of the image, use

```
redPlane = RGB(:,:,1);
```

To access a subimage, use

```
subimage = RGB(20:40,50:85,:);
```

The RGB image is a good example of data that needs to be accessed in planes for operations like display or filtering. In other instances, however, the data itself might be multidimensional. For example, consider a set of temperature measurements taken at equally spaced points in a room. Here the location of each value is an integral part of the data set—the physical placement in three-space of each element is an aspect of the information. Such data also lends itself to representation as a multidimensional array.



Now to find the average of all the measurements, use

```
mean(mean(mean(TEMP)));
```

To obtain a vector of the "middle" values (element (2,2)) in the room on each page, use

```
B = TEMP(2,2,:);
```

Multidimensional Cell Arrays

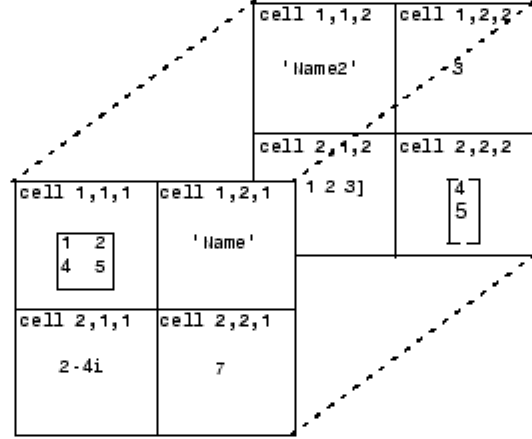
Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the `cat` function to build multidimensional cell arrays, just as you use it for numeric arrays.

For example, create a simple three-dimensional cell array `C`:

```
A{1,1} = [1 2;4 5];
A{1,2} = 'Name';
A{2,1} = 2-4i;
```

```
A{2,2} = 7;
B{1,1} = 'Name2';
B{1,2} = 3;
B{2,1} = 0:1:3;
B{2,2} = [4 5]';
C = cat(3, A, B);
```

The subscripts for the cells of C look like

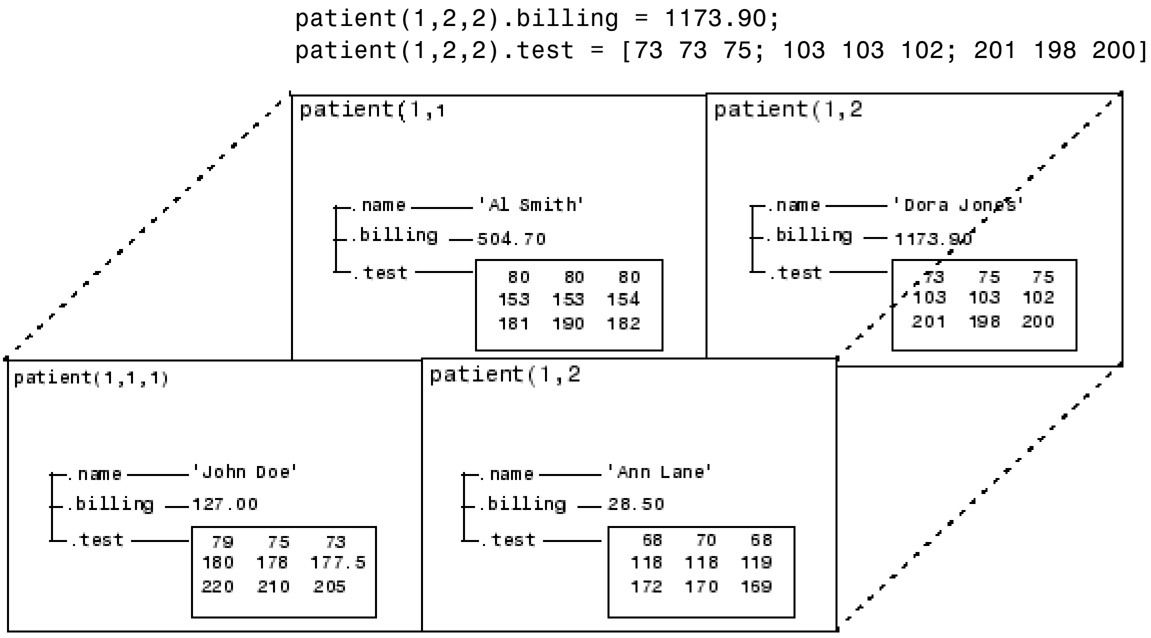


Multidimensional Structure Arrays

Multidimensional structure arrays are extensions of rectangular structure arrays. Like other types of multidimensional arrays, you can build them using direct assignment or the cat function:

```
patient(1,1,1).name = 'John Doe';
patient(1,1,1).billing = 127.00;
patient(1,1,1).test = [79 75 73; 180 178 177.5; 220 210 205];
patient(1,2,1).name = 'Ann Lane';
patient(1,2,1).billing = 28.50;
patient(1,2,1).test = [68 70 68; 118 118 119; 172 170 169];
patient(1,1,2).name = 'Al Smith';
patient(1,1,2).billing = 504.70;
patient(1,1,2).test = [80 80 80; 153 153 154; 181 190 182];
patient(1,2,2).name = 'Dora Jones';
```

```
patient(1,2,2).billing = 1173.90;
patient(1,2,2).test = [73 73 75; 103 103 102; 201 198 200];
```



Applying Functions to Multidimensional Structure Arrays

To apply functions to multidimensional structure arrays, operate on fields and field elements using indexing. For example, find the sum of the columns of the `test` array in `patient(1,1,2)`:

```
sum((patient(1,1,2).test));
```

Similarly, add all the `billing` fields in the `patient` array:

```
total = sum([patient.billing]);
```

Summary of Matrix and Array Functions

This section summarizes the principal functions used in creating and handling matrices. Most of these functions work on multidimensional arrays as well.

Functions to Create a Matrix

Function	Description
`[a,b]` or `[a;b]`	Create a matrix from specified elements, or concatenate matrices together.
`accumarray`	Construct a matrix using accumulation.
`blkdiag`	Construct a block diagonal matrix.
`cat`	Concatenate matrices along the specified dimension.
`diag`	Create a diagonal matrix from a vector.
`horzcat`	Concatenate matrices horizontally.
`magic`	Create a square matrix with rows, columns, and diagonals that add up to the same number.
`ones`	Create a matrix of all ones.
`rand`	Create a matrix of uniformly distributed random numbers.
`repmat`	Create a new matrix by replicating or tiling another.
`vertcat`	Concatenate two or more matrices vertically.
`zeros`	Create a matrix of all zeros.

Functions to Modify the Shape of a Matrix

Function	Description
ctranspose	Flip a matrix about the main diagonal and replace each element with its complex conjugate.
`flipdim`	Flip a matrix along the specified dimension.
`fliplr`	Flip a matrix about a vertical axis.

Functions to Modify the Shape of a Matrix (Continued)

Function	Description
flipud	Flip a matrix about a horizontal axis.
reshape	Change the dimensions of a matrix.
rot90	Rotate a matrix by 90 degrees.
transpose	Flip a matrix about the main diagonal.

Functions to Find the Structure or Shape of a Matrix

Function	Description
isempty	Return true for 0-by-0 or 0-by-n matrices.
isscalar	Return true for 1-by-1 matrices.
issparse	Return true for sparse matrices.
isvector	Return true for 1-by-n matrices.
length	Return the length of a vector.
ndims	Return the number of dimensions in a matrix.
numel	Return the number of elements in a matrix.
size	Return the size of each dimension.

Functions to Determine Data Type

Function	Description
iscell	Return true if the matrix is a cell array.
ischar	Return true if matrix elements are characters or strings.
isfloat	Determine if input is a floating point array.
isinteger	Determine if input is an integer array.
islogical	Return true if matrix elements are logicals.
isnumeric	Return true if matrix elements are numeric.

Functions to Determine Data Type (Continued)

Function	Description
isreal	Return true if matrix elements are real numbers.
isstruct	Return true if matrix elements are MATLAB structures.

Functions to Sort and Shift Matrix Elements

Function	Description
circshift	Circularly shift matrix contents.
issorted	Return true if the matrix elements are sorted.
sort	Sort elements in ascending or descending order.
sortrows	Sort rows in ascending order.

Functions That Work on Diagonals of a Matrix

Function	Description
blkdiag	Construct a block diagonal matrix.
diag	Return the diagonals of a matrix.
trace	Compute the sum of the elements on the main diagonal.

Functions That Work on Diagonals of a Matrix (Continued)

Function	Description
`tril`	Return the lower triangular part of a matrix.
`triu`	Return the upper triangular part of a matrix.

Functions to Change the Indexing Style

Function	Description
`ind2sub`	Convert a linear index to a row-column index.
`sub2ind`	Convert a row-column index to a linear index.

Functions for Working with Multidimensional Arrays

Function	Description
`cat`	Concatenate arrays.
`circshift`	Shift array circularly.
`ipermute`	Inverse permute array dimensions.
`ndgrid`	Generate arrays for `n`-dimensional functions and interpolation.
`ndims`	Return the number of array dimensions.
`permute`	Permute array dimensions.
`shiftdim`	Shift array dimensions.
`squeeze`	Remove singleton dimensions.

2

Data Types

Overview of MATLAB Data Types (p. 2-3)	Brief description of all MATLAB data types
Numeric Types (p. 2-6)	Integer and floating-point data types, complex numbers, `NaN`, infinity, and numeric display format
Logical Types (p. 2-33)	States of `true` and `false`, use of logicals in conditional statements and logical indexing, logical/numeric conversion
Characters and Strings (p. 2-37)	Characters, strings, cell arrays of strings, string comparison, search and replace, character/numeric conversion
Dates and Times (p. 2-66)	Date strings, serial date numbers, date vectors, date type conversion, output display format
Structures (p. 2-74)	C-like structures with named fields, dynamic field names, adding and removing fields
Cell Arrays (p. 2-93)	Arrays of cells containing different data types and shapes, using cell arrays in argument lists, numeric/cell conversion
Function Handles (p. 2-115)	Passing function access data to other functions, extending function scope, extending the lifetime of variables

MATLAB Classes (p. 2-117)	Object-oriented classes and methods using MATLAB classes, creating your own MATLAB data types
Java Classes (p. 2-118)	Working with Java classes within MATLAB using the MATLAB interface to the Java programming language

Overview of MATLAB Data Types

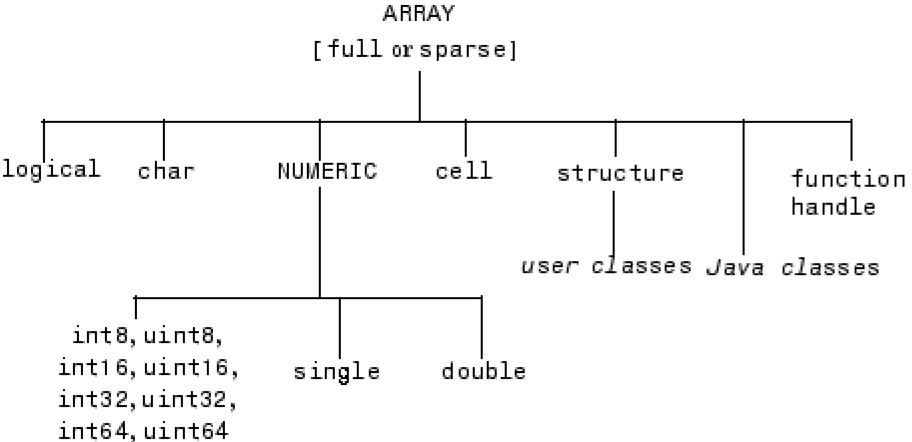
In this section...
"Fundamental Data Types" on page 2-3
"How to Use the Different Types" on page 2-4

Fundamental Data Types

There are many different types of data that you can work with in MATLAB. You can build matrices and arrays of floating-point and integer data, characters and strings, logical `true` and `false` states, etc. Two of the MATLAB data types, structures and cell arrays, provide a way to store dissimilar types of data in the same array. You can also develop your own data types using MATLAB classes.

There are 15 fundamental data types in MATLAB. Each of these data types is in the form of a matrix or array. This matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size.

All of the fundamental data types are shown in lowercase, plain nonitalic text in the diagram below.



The two data types shown in italic text are user-defined, object-oriented *`user classes`* and *`Java classes`*. You can use the latter with the MATLAB interface to Java (see "Calling Java from MATLAB" in the MATLAB External Interfaces documentation).

You can create two-dimensional `double` and `logical` matrices using one of two storage formats: `full` or `sparse`. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These data types require different amounts of storage, the smallest being a `logical` value or 8–bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

How to Use the Different Types

The following table describes these data types in more detail.

Data Type	Example	Description
`int8, uint8, int16, uint16, int32, uint32, int64, uint64`	`uint16(65000)`	Array of signed (`int`) and unsigned (`uint`) integers. Some integer types require less storage space than `single` or `double`. All integer types except for `int64` and `uint64` can be used in mathematical operations.
`single`	`single(3 * 10^38)`	Array of single-precision numbers. Requires less storage space than `double`, but has less precision and a smaller range.
`double`	`3 * 10^300` `5 + 6i`	Array of double-precision numbers. Two-dimensional arrays can be sparse. The default numeric type in MATLAB.
`logical`	`magic(4) > 10`	Array of logical values of `1` or `0` to represent `true` and `false` respectively. Two-dimensional arrays can be sparse.

Data Type	Example	Description
`char`	`'Hello'`	Array of characters. Strings are represented as vectors of characters. For arrays containing more than one string, it is best to use cell arrays.
`cell array`	`a{1,1} = 12; a{1,2} = 'Red'; a{1,3} = magic(4);`	Array of indexed cells, each capable of storing an array of a different dimension and data type.
`structure`	`a.day = 12; a.color = 'Red'; a.mat = magic(3);`	Array of C-like structures, each structure having named fields capable of storing an array of a different dimension and data type.
`function handle`	`@sin`	Pointer to a function. You can pass function handles to other functions.
`user class`	`polynom([0 -2 -5])`	Objects constructed from a user-defined class. See "MATLAB Classes" on page 2-117
`Java class`	`java.awt.Frame`	Objects constructed from a Java class. See "Java Classes" on page 2-118.

Numeric Types

In this section...

"Overview" on page 2-6
"Integers" on page 2-6
"Floating-Point Numbers" on page 2-14
"Complex Numbers" on page 2-24
"Infinity and NaN" on page 2-25
"Identifying Numeric Types" on page 2-27
"Display Format for Numeric Values" on page 2-27
"Function Summary" on page 2-29

Overview

Numeric data types in MATLAB include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting and reshaping. All numeric types except for `int64` and `uint64` can be used in mathematical operations.

Integers

MATLAB has four signed and four unsigned integer data types. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

This section covers:

- "Creating Integer Data" on page 2-7
- "Arithmetic Operations on Integer Data Types" on page 2-9
- "Largest and Smallest Values for Integer Data Types" on page 2-9
- "Warnings for Integer Data Types" on page 2-10
- "Integer Functions" on page 2-13

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you don't need a 32-bit integer to store the value 100.

Here are the eight integer data types, the range of values you can store with each type, and the MATLAB conversion function required to create that type:

Data Type	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	int8
Signed 16-bit integer	-2^{15} to 2^{15}-1	int16
Signed 32-bit integer	-2^{31} to 2^{31}-1	int32
Signed 64-bit integer	-2^{63} to 2^{63}-1	int64
Unsigned 8-bit integer	0 to 2^8-1	uint8
Unsigned 16-bit integer	0 to 2^{16}-1	uint16
Unsigned 32-bit integer	0 to 2^{32}-1	uint32
Unsigned 64-bit integer	0 to 2^{64}-1	uint64

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (double) by default. To store data as an integer, you need to convert from double to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store 325 as a 16-bit signed integer assigned to variable x, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude:

```
x = 325.499;                    x = x + .001;

int16(x)                        int16(x)
ans =                           ans =
     325                             326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. In this example, the `fix` function enables you to override the default and round *towards zero* when the fractional part of a number is `.5`:

```
x = 325.5;

int16(fix(x))
ans =
     325
```

Arithmetic operations that involve both integers and floating-point always result in an integer data type. MATLAB rounds the result, when necessary, according to the default rounding algorithm. The example below yields an exact answer of `1426.75` which MATLAB then rounds to the next highest integer:

```
int16(325) * 4.39
ans =
     1427
```

The integer conversion functions are also useful when converting other data types, such as strings, to integers:

```
str = 'Hello World';

int8(str)
ans =
   72  101  108  108  111   32   87  111  114  108  100
```

Arithmetic Operations on Integer Data Types

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. This yields a result that has the same data type as the operands:

  ```
  x = uint32([132 347 528]) .* uint32(75);
  ```

- Integers or integer arrays and scalar double-precision floating-point numbers. This yields a result that has the same data type as the integer operands:

  ```
  x = uint32([132 347 528]) .* 75.49;
  ```

For all binary operations in which one operand is an array of integer data type and the other is a scalar `double`, MATLAB computes the operation using elementwise double-precision arithmetic, and then converts the result back to the original integer data type.

For a list of the operations that support integer data types, see Nondouble Data Type Support in the arithmetic operators reference page.

Largest and Smallest Values for Integer Data Types

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under "Integers" on page 2-6 lists the largest and smallest values for each integer data type in the "Range of Values" column.

You can also obtain these values with the `intmax` and `intmin` functions:

```
intmax('int8')                intmin('int8')
ans =                         ans =
   127                           -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if

you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example,

```
x = int8(300)                x = int8(-300)
x =                          x =
   127                          -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100) * 3            x = int8(-100) * 3
x =                          x =
   127                          -128
```

You can make MATLAB return a warning when your input is outside the range an integer data type. This is described in the next section.

Warnings for Integer Data Types

Use the `intwarning` function to make MATLAB return a warning message when it converts a number outside the range of an integer data type to that type, or when the result of an arithmetic operation overflows. There are four possible warning messages that you can turn on using `intwarning`:

Message Identifier	Reason for Warning
`MATLAB:intConvertOverflow`	Overflow when attempting to convert from a numeric class to an integer class
`MATLAB:intMathOverflow`	Overflow when attempting an integer arithmetic operation
`MATLAB:intConvertNonIntVal`	Attempt to convert a noninteger value to an integer
`MATLAB:intConvertNaN`	Attempt to convert NaN (Not a Number) to an integer

Querying the Present Warning State. Use the following command to display the state of all integer warnings:

```
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'off'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
   The state of warning 'MATLAB:intConvertOverflow' is 'off'.
   The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

To display the state of one or more selected warnings, index into the structure returned by `intwarning`. This example shows the current state of the `intConvertOverflow` warning:

```
iwState = intwarning('query');
iwState(3)
ans =
    identifier: 'MATLAB:intConvertOverflow'
         state: 'off'
```

Turning the Warning On. To enable all four integer warnings, use `intwarning` with the string `'on'`:

```
intwarning('on');
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'on'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
   The state of warning 'MATLAB:intConvertOverflow' is 'on'.
   The state of warning 'MATLAB:intMathOverflow' is 'on'.
```

To enable one or more selected integer warnings, first make sure that all integer warnings are disabled:

```
intwarning('off');
```

Note that, in this state, the following conversion to a 16-bit integer overflows, but does not issue a warning:

```
x = int16(50000)
x =
  32767
```

Find which of the four warnings covers integer conversion. In this case, it is the third in the structure array:

```
iwState = intwarning('query');
iwState(3).identifier
ans =
   MATLAB:intConvertOverflow
```

Set the warning state to `'on'` in the structure, and then call `intwarning` using the structure as input:

```
iwState(3).state = 'on';
intwarning(iwState);
```

With the warning enabled, the overflow on conversion does issue the warning message:

```
x = int16(50000)
Warning: Out of range value converted to intmin('int16') or
intmax('int16').
x =
  32767
```

You can also use the following `for` loop to enable integer warnings selectively:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, 'MATLAB:intConvertOverflow')
      iwState(k).state = 'on';
      intwarning(iwState);
   end
end
```

Turning the Warning Off. To turn all integer warnings off (their default state when you start MATLAB), enter

```
intwarning('off')
```

To disable selected integer warnings, follow the steps shown for enabling warnings, but with the `state` field of the structure set to `'off'`:

```
iwState(3).identifier
ans =
```

```
   MATLAB:intConvertOverflow

iwState(3).state = 'off';
intwarning(iwState);
```

Turning Warnings On or Off Temporarily. When writing M-files that contain integer data types, it is sometimes convenient to temporarily turn integer warnings on, and then return the states of the warnings (`'on'` or `'off'`) to their previous settings. The following commands illustrate how to do this:

```
oldState = intwarning('on');

int8(200);
Warning: Out of range value converted to intmin('int8') or
intmax('int8').

intwarning(oldState)
```

To temporarily turn the warnings off, change the first line of the preceding code to

```
oldState = intwarning('off');
```

Recommended Usage of Math Overflow Warning. Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed above do not affect program performance.

Note that calling `intwarning('on')` enables all four integer warnings, including the `intMathOverflow` warning that can have an effect on integer arithmetic.

Integer Functions

See Integer Functions on page 2-30 for a list of functions most commonly used with integers in MATLAB.

Floating-Point Numbers

MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

This section covers:

- "Double-Precision Floating Point" on page 2-14
- "Single-Precision Floating Point" on page 2-15
- "Creating Floating-Point Data" on page 2-15
- "Arithmetic Operations on Floating-Point Numbers" on page 2-17
- "Largest and Smallest Values for Floating-Point Data Types" on page 2-18
- "Accuracy of Floating-Point Data" on page 2-19
- "Avoiding Common Problems with Floating-Point Arithmetic" on page 2-21
- "Floating-Point Functions" on page 2-23
- "References" on page 2-23

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (0 = positive, 1 = negative)
62 to 52	Exponent, biased by 1023
51 to 0	Fraction `f` of the number `1.f`

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction f of the number 1.f

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38}. For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Double-Precision. Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

```
whos(x)
  Name      Size                       Bytes  Class

  x         1x1                            8  double
```

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (`true`) if the input is a floating-point number, and logical 0 (`false`) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, `double`. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = double(y)                      % Convert to double
x =
   -5.8932e+011
```

Single-Precision. Because MATLAB stores numeric data as a `double` by default, you need to use the `single` conversion function to create a single-precision number:

```
x = single(25.783);
```

The `whos` function returns the attributes of variable `x` in a structure. The `bytes` field of this structure shows that when `x` is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a `double`:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the `single` function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = single(y)                      % Convert to single
x =
   -5.8932e+011
```

Arithmetic Operations on Floating-Point Numbers

This section describes which data types you can use in arithmetic operations with floating-point numbers.

Double-Precision. You can perform basic arithmetic operations with `double` and any of the following other data types. When one or more operands is an integer (scalar or array), the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise:

- `single` — The result is of type `single`
- `double`
- `int*` or `uint*` — The result has the same data type as the integer operand
- `char`
- `logical`

This example performs arithmetic on data of types `char` and `double`. The result is of type `double`:

```
c = 'uppercase' - 32;

class(c)
ans =
   double

char(c)
ans =
   UPPERCASE
```

Single-Precision. You can perform basic arithmetic operations with `single` and any of the following other data types. The result is always `single`:

- `single`
- `double`
- `char`
- `logical`

In this example, 7.5 defaults to type double, and the result is of type single:

```
x = single([1.32 3.47 5.28]) .* 7.5;

class(x)
ans =
   single
```

Largest and Smallest Values for Floating-Point Data Types

For the double and single data types, there is a largest and smallest number that you can represent with that type.

Double-Precision. The MATLAB functions realmax and realmin return the maximum and minimum values that you can represent with the double data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax, -realmin, realmin, realmax)

ans =
The range for double is:
   -1.79769e+308 to -2.22507e-308 and
    2.22507e-308 to  1.79769e+308
```

Numbers larger than realmax or smaller than -realmax are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
ans =
   Inf

-realmax - .0001e+308
ans =
  -Inf
```

Single-Precision. The MATLAB functions realmax and realmin, when called with the argument 'single', return the maximum and minimum values that you can represent with the single data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax('single'), -realmin('single'), ...
   realmin('single'), realmax('single'))

ans =
The range for single is:
   -3.40282e+038 to -1.17549e-038 and
    1.17549e-038 to  3.40282e+038
```

Numbers larger than `realmax`('single') or smaller than `-realmax` ('single') are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
ans =
   Inf

-realmax('single') - .0001e+038
ans =
  -Inf
```

Accuracy of Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as you had expected, it is likely caused by the limitations of your computer's hardware. Probably, your result was a little less exact because the hardware had insufficient bits to represent the result with perfect accuracy; therefore, it truncated the resulting value.

Double-Precision. Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the `eps` function. For example, to find the distance between `5` and the next larger double-precision number, enter

```
format long

eps(5)
ans =
```

```
        8.881784197001252e-016
```

This tells you that there are no double-precision numbers between 5 and `5 + eps(5)`. If a double-precision computation returns the answer 5, the result is only accurate to within `eps(5)`.

The value of `eps(x)` depends on `x`. This example shows that, as `x` gets larger, so does `eps(x)`:

```
eps(50)
ans =
    7.105427357601002e-015
```

If you enter `eps` with no input argument, MATLAB returns the value of `eps(1)`, the distance from `1` to the next larger double-precision number.

Single-Precision. Similarly, there are gaps between any two single-precision numbers. If `x` has type `single`, `eps(x)` returns the distance between `x` and the next larger single-precision number. For example,

```
x = single(5);
eps(x)
```

returns

```
ans =
  4.7684e-007
```

Note that this result is larger than `eps(5)`. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number `x` of type `double`, `eps(single(x))` gives you an upper bound for the amount that `x` is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number `3.14` to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
```

```
  1.0490e-007
```

The amount that `3.14` is rounded is less than

```
eps(single(3.14))
ans =
  2.3842e-007
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

Example 1 — Round-Off or What You Get Is Not What You Expect.

The decimal number `4/3` is not exactly representable as a binary fraction. For this reason, the following calculation does not give zero, but rather reveals the quantity `eps`.

```
e = 1 - 3*(4/3 - 1)

e =
  2.2204e-016
```

Similarly, `0.1` is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a == 1

ans =
     0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c

ans =
     0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53

ans =
     0
```

Since `pi` is not really `pi`, it is not surprising that `sin(pi)` is not exactly zero:

```
sin(pi)

ans =
   1.224646799147353e-016
```

Example 2 — Catastrophic Cancellation. When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant):

```
sqrt(1e-16 + 1) - 1

ans =
     0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

Example 3 — Floating-Point Operations and Linear Algebra. Round-off, cancellation, and other traits of floating-point arithmetic combine to produce startling computations when solving the problems of linear algebra. MATLAB warns that the following matrix `A` is ill-conditioned, and therefore the system `Ax = b` may be sensitive to small perturbations. Although the computation differs from what you expect in exact arithmetic, the result is correct.

```
A = [2 eps -eps; eps 1 1; -eps 1 1];
b = [2; eps + 2; -eps + 2];
x = A\b

x =
   1.0e+015 *
    0.000000000000001
    2.251799813685249
   -2.251799813685247
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB. For more examples and information, see `Technical Note 1108   Common Problems with Floating-Point Arithmetic`.

Floating-Point Functions

See Floating-Point Functions on page 2-30 for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

[1] Moler, Cleve, "Floating Points," *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/moler/.

Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of -1. This is represented in MATLAB by either of two letters: i or j.

Creating Complex Numbers

The following statement shows one way of creating a complex value in MATLAB. The variable x is assigned a complex number with a real part of 2 and an imaginary part of 3:

```
x = 2 + 3i;
```

Another way to create a complex number is using the complex function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = rand(3) * 5;
y = rand(3) * -8;

z = complex(x, y)
z =
   4.7842 -1.0921i   0.8648 -1.5931i   1.2616 -2.2753i
   2.6130 -0.0941i   4.8987 -2.3898i   4.3787 -3.7538i
   4.4007 -7.1512i   1.3572 -5.2915i   3.6865 -0.5182i
```

You can separate a complex number into its real and imaginary parts using the real and imag functions:

```
zr = real(z)
zr =
    4.7842    0.8648    1.2616
    2.6130    4.8987    4.3787
    4.4007    1.3572    3.6865

zi = imag(z)
zi =
   -1.0921   -1.5931   -2.2753
   -0.0941   -2.3898   -3.7538
   -7.1512   -5.2915   -0.5182
```

Complex Number Functions

See Complex Number Functions on page 2-31 for a list of functions most commonly used with MATLAB complex numbers in MATLAB.

Infinity and NaN

MATLAB uses the special values `inf`, `-inf`, and `NaN` to represent values that are positive and negative infinity, and not a number respectively.

Infinity

MATLAB represents infinity by the special value `inf`. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values. MATLAB also provides a function called `inf` that returns the IEEE arithmetic representation for positive infinity as a `double` scalar value.

Several examples of statements that return positive or negative infinity in MATLAB are shown here.

x = 1/0 x = Inf	x = 1.e1000 x = Inf
x = exp(1000) x = Inf	x = log(0) x = -Inf

Use the `isinf` function to verify that `x` is positive or negative infinity:

```
x = log(0);

isinf(x)
ans =
     1
```

NaN

MATLAB represents values that are not real or complex numbers with a special value called NaN, which stands for Not a Number. Expressions like 0/0 and inf/inf result in NaN, as do any arithmetic operations involving a NaN.

For example, the statement n/0, where n is complex, returns NaN for the real part of the result:

```
x = 7i/0
x =
      NaN +    Infi
```

Use the isnan function to verify that the real part of x is NaN:

```
isnan(real(x))
ans =
     1
```

MATLAB also provides a function called NaN that returns the IEEE arithmetic representation for NaN as a double scalar value:

```
x = NaN;

whos x
  Name      Size                    Bytes  Class

  x         1x1                         8  double
```

Logical Operations on NaN. Because two NaNs are not equal to each other, logical operations involving NaN always return false, except for a test for inequality, (NaN ~= NaN):

```
NaN > NaN
ans =
     0

NaN ~= NaN
ans =
     1
```

Infinity and NaN Functions

See Infinity and NaN Functions on page 2-31 for a list of functions most commonly used with `inf` and `NaN` in MATLAB.

Identifying Numeric Types

You can check the data type of a variable `x` using any of these commands.

Command	Operation
`whos x`	Display the data type of `x`.
`xType = class(x);`	Assign the data type of `x` to a variable.
`isnumeric(x)`	Determine if `x` is a numeric type.

`isa(x, 'integer')`
`isa(x, 'uint64')`
`isa(x, 'float')`
`isa(x, 'double')`
`isa(x, 'single')`
	Determine if `x` is the specified numeric type. (Examples for any integer, unsigned 64-bit integer, any floating point, double precision, and single precision are shown here).

`isreal(x)`	Determine if `x` is real or complex.
`isnan(x)`	Determine if `x` is Not a Number (`NaN`).
`isinf(x)`	Determine if `x` is infinite.
`isfinite(x)`	Determine if `x` is finite.

Display Format for Numeric Values

By default, MATLAB displays numeric output as 5-digit scaled, fixed-point values. You can change the way numeric values are displayed to any of the following:

- 5-digit scaled fixed point, floating point, or the best of the two
- 15-digit scaled fixed point, floating point, or the best of the two
- A ratio of small integers
- Hexadecimal (base 16)
- Bank notation

All available formats are listed on the `format` reference page.

To change the numeric display setting, use either the `format` function or the **Preferences** dialog box (accessible from the MATLAB **File** menu). The `format` function changes the display of numeric values for the duration of a single MATLAB session, while your Preferences settings remain active from one session to the next. These settings affect only how numbers are displayed, not how MATLAB computes or saves them.

Display Format Examples

Here are a few examples of the various formats and the output produced from the following two-element vector `x`, with components of different magnitudes.

Check the current format setting:

```
get(0, 'format')
ans =
   short
```

Set the value for `x` and display in 5-digit scaled fixed point:

```
x = [4/3 1.2345e-6]
x =
    1.3333    0.0000
```

Set the format to 5-digit floating point:

```
format short e
x
x =
  1.3333e+000  1.2345e-006
```

Set the format to 15-digit scaled fixed point:

```
format long
x
x =
   1.33333333333333   0.00000123450000
```

Set the format to `'rational'` for small integer ratio output:

```
format rational
x
x =
      4/3          1/810045
```

Set an integer value for x and display it in hexadecimal (base 16) format:

```
format hex
x = uint32(876543210)
x =
   343efcea
```

Setting Numeric Format in a Program

To temporarily change the numeric format inside a program, get the original format using the get function and save it in a variable. When you finish working with the new format, you can restore the original format setting using the set function as shown here:

```
origFormat = get(0, 'format');
format('rational');

  -- Work in rational format --

set(0,'format', origFormat);
```

Function Summary

MATLAB provides these functions for working with numeric data types:

- Integer Functions on page 2-30
- Floating-Point Functions on page 2-30
- Complex Number Functions on page 2-31
- Infinity and NaN Functions on page 2-31
- Type Identification Functions on page 2-32
- Output Formatting Functions on page 2-32

Integer Functions

Function	Description
`int8, int16, int32, int64`	Convert to signed 1-, 2-, 4-, or 8-byte integer.
`uint8, uint16, uint32, uint64`	Convert to unsigned 1-, 2-, 4-, or 8-byte integer.
`ceil`	Round towards plus infinity to nearest integer
`class`	Return the data type of an object.
`fix`	Round towards zero to nearest integer
`floor`	Round towards minus infinity to nearest integer
`isa`	Determine if input value has the specified data type.
`isinteger`	Determine if input value is an integer array.
`isnumeric`	Determine if input value is a numeric array.
`round`	Round towards the nearest integer

Floating-Point Functions

Function	Description
`double`	Convert to double precision.
`single`	Convert to single precision.
`class`	Return the data type of an object.
`isa`	Determine if input value has the specified data type.
`isfloat`	Determine if input value is a floating-point array.
`isnumeric`	Determine if input value is a numeric array.
`eps`	Return the floating-point relative accuracy. This value is the tolerance MATLAB uses in its calculations.

Floating-Point Functions (Continued)

Function	Description
realmax	Return the largest floating-point number your computer can represent.
realmin	Return the smallest floating-point number your computer can represent.

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components.
i or j	Return the imaginary unit used in constructing complex data.
real	Return the real part of a complex number.
imag	Return the imaginary part of a complex number.
isreal	Determine if a number is real or imaginary.

Infinity and NaN Functions

Function	Description
inf	Return the IEEE value for infinity.
isnan	Detect NaN elements of an array.
isinf	Detect infinite elements of an array.

Infinity and NaN Functions (Continued)

Function	Description
isfinite	Detect finite elements of an array.
nan	Return the IEEE value for Not a Number.

Type Identification Functions

Function	Description
class	Return data type (or class).
isa	Determine if input value is of the specified data type.
isfloat	Determine if input value is a floating-point array.
isinteger	Determine if input value is an integer array.
isnumeric	Determine if input value is a numeric array.
isreal	Determine if input value is real.
whos	Display the data type of input.

Output Formatting Functions

Function	Description
format	Control display format for output.

Logical Types

In this section...

"Overview" on page 2-33

"Creating a Logical Array" on page 2-33

"How Logical Arrays Are Used" on page 2-35

"Identifying Logical Arrays" on page 2-36

Overview

The `logical` data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `true` or `false` to indicate whether a certain condition was found to be true or not. For example, the statement `(5 * 10) > 40` returns a logical `true` value.

Logical data does not have to be scalar; MATLAB supports arrays of logical values as well. For example, the following statement returns a vector of logicals indicating `false` for the first two elements and `true` for the last three:

```
[30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

Creating a Logical Array

One way of creating an array of logicals is to just enter a `true` or `false` value for each element. The `true` function returns logical one; the `false` function returns logical zero:

```
x = [true, true, false, true, false];
```

Logical Operations on an Array

You can also perform some logical operation on an array that yields an array of logicals:

```
x = magic(4) >= 9
```

```
x =
     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The MATLAB functions that have names beginning with `is` (e.g., `ischar`, `issparse`) also return a logical value or array:

```
a = [2.5 6.7 9.2 inf 4.8];

isfinite(a)
ans =
     1     1     1     0     1
```

This table shows some of the MATLAB operations that return a logical `true` or `false`.

<table>
<tr><th>Function</th><th>Operation</th></tr>
<tr><td><code>true</code>, <code>false</code></td><td>Setting value to <code>true</code> or <code>false</code></td></tr>
<tr><td><code>logical</code></td><td>Numeric to logical conversion</td></tr>
<tr><td>& (<code>and</code>), | (<code>or</code>), ~ (<code>not</code>), <code>xor</code>, <code>any</code>, <code>all</code></td><td>Logical operations</td></tr>
<tr><td>&&, ||</td><td>Short-circuit AND and OR</td></tr>
<tr><td>== (<code>eq</code>), ~= (<code>ne</code>), < (<code>lt</code>), > (<code>gt</code>), <= (<code>le</code>), >= (<code>ge</code>)</td><td>Relational operations</td></tr>
<tr><td>All <code>is*</code> functions, <code>cellfun</code></td><td>Test operations</td></tr>
<tr><td><code>strcmp</code>, <code>strncmp</code>, <code>strcmpi</code>, <code>strncmpi</code></td><td>String comparisons</td></tr>
</table>

Sparse Logical Arrays

Logical arrays can also be sparse as long as they have no more than two dimensions:

```
x = sparse(magic(20) > 395)
x =
   (1,1)        1
   (1,4)        1
```

```
(1,5)        1
(20,18)      1
(20,19)      1
```

How Logical Arrays Are Used

MATLAB has two primary uses for logical arrays:

- "Using Logicals in Conditional Statements" on page 2-35
- "Logical Indexing" on page 2-35

Most mathematics operations are not supported on logical values.

Using Logicals in Conditional Statements

Conditional statements are useful when you want to execute a block of code only when a certain condition is met. For example, the `sprintf` command shown below is valid only if `str` is a nonempty string. The statement

```
if ~isempty(str) && ischar(str)
```

checks for this condition and allows the `sprintf` to execute only if it is `true`:

```
str = 'Hello';

if ~isempty(str) && ischar(str)
   sprintf('Input string is ''%s''', str)
   end

ans =
   Input string is 'Hello'
```

Logical Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See “Using Logicals in Array Indexing” on page 1-22 for more information on this subject.

Identifying Logical Arrays

This table shows the commands you can use to determine whether or not an array `x` is logical. The last function listed, `cellfun`, operates on cell arrays, which you can read about in the section “Cell Arrays” on page 2-93.

Command	Operation
`whos(x)`	Display value and data type for `x`.
`islogical(x)`	Return `true` if array is logical.
`isa(x, 'logical')`	Return `true` if array is logical.
`class(x)`	Return string with data type name.
`cellfun('islogical', x)`	Check each cell array element for logical.

Characters and Strings

In this section...

"Overview" on page 2-37

"Creating Character Arrays" on page 2-37

"Cell Arrays of Strings" on page 2-39

"Formatting Strings" on page 2-42

"String Comparisons" on page 2-55

"Searching and Replacing" on page 2-58

"Converting from Numeric to String" on page 2-59

"Converting from String to Numeric" on page 2-61

"Function Summary" on page 2-63

Overview

In MATLAB, the term *string* refers to an array of Unicode characters. MATLAB represents each character internally as its corresponding numeric value. Unless you want to access these values, you can simply work with the characters as they display on screen.

You can use `char` to hold an `m`-by-`n` array of strings as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To hold an array of strings of unequal length, use a cell array.

The string is actually a vector whose components are the numeric codes for the characters. The actual characters displayed depend on the character set encoding for a given font.

Creating Character Arrays

Specify character data by placing characters inside a pair of single quotes. For example, this line creates a 1-by-13 character array called `name`:

```
name = 'Thomas R. Lee';
```

In the workspace, the output of whos shows

```
Name        Size           Bytes  Class

name        1x13              26  char
```

You can see that each character uses 2 bytes of storage internally.

The class and ischar functions show that name is a character array:

```
class(name)
ans =
   char

ischar(name)
ans =
    1
```

You also can join two or more character arrays together to create a new character array. To do this, use either the string concatenation function, strcat, or the MATLAB concatenation operator, []. The latter preserves any trailing spaces found in the input arrays:

```
name = 'Thomas R. Lee';
title = ' Sr. Developer';

strcat(name,',',title)
ans =
   Thomas R. Lee, Sr. Developer
```

To concatenate strings vertically, use strvcat.

Creating Two-Dimensional Character Arrays

When creating a two-dimensional character array, be sure that each row has the same length. For example, this line is legal because both input rows have exactly 13 characters:

```
name = ['Thomas R. Lee' ; 'Sr. Developer']
name =
   Thomas R. Lee
```

```
    Sr. Developer
```

When creating character arrays from strings of different lengths, you can pad the shorter strings with blanks to force rows of equal length:

```
name = ['Thomas R. Lee   '; 'Senior Developer'];
```

A simpler way to create string arrays is to use the `char` function. `char` automatically pads all strings to the length of the longest input string. In the following example, `char` pads the 13-character input string `'Thomas R. Lee'` with three trailing blanks so that it will be as long as the second string:

```
name = char('Thomas R. Lee','Senior Developer')
name =
   Thomas R. Lee
   Senior Developer
```

When extracting strings from an array, use the `deblank` function to remove any trailing blanks:

```
trimname = deblank(name(1,:))
trimname =
   Thomas R. Lee

size(trimname)
ans =
     1    13
```

Expanding Character Arrays

Expanding the size of an existing character array by assigning additional characters to indices beyond the bounds of the array such that part of the array becomes padded with zeros, is generally not recommended. See the documentation on “Expanding a Character Array” on page 1-35 in the MATLAB Programming documentation.

Cell Arrays of Strings

Creating strings in a regular MATLAB array requires that all strings in the array be of the same length. This often means that you have to pad blanks at

the end of strings to equalize their length. However, another type of MATLAB array, the cell array, can hold different sizes and types of data in an array without padding. Cell arrays provide a more flexible way to store strings of varying length.

For details on cell arrays, see “Cell Arrays” on page 2-93.

Converting to a Cell Array of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider the character array

```
data = ['Allison Jones';'Development  ';'Phoenix      '];
```

Each row of the matrix is padded so that all have equal length (in this case, 13 characters).

Now use `cellstr` to create a column vector of cells, each cell containing one of the strings from the `data` array:

```
celldata = cellstr(data)
celldata =
    'Allison Jones'
    'Development'
    'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix:

```
length(celldata{3})
ans =
    7
```

The `iscellstr` function determines if the input argument is a cell array of strings. It returns a logical `1` (`true`) in the case of `celldata`:

```
iscellstr(celldata)
ans =
     1
```

Use `char` to convert back to a standard padded character array:

```
strings = char(celldata)
strings =
   Allison Jones
   Development
   Phoenix

length(strings(3,:))
ans =
    13
```

Functions for Cell Arrays of Strings

This table describes the MATLAB functions for working with cell arrays.

Function	Description
`cellstr`	Convert a character array to a cell array of strings.
`char`	Convert a cell array of strings to a character array.
`deblank`	Remove trailing blanks from a string.
`iscellstr`	Return `true` for a cell array of strings.
`sort`	Sort elements in ascending or descending order.
`strcat`	Concatenate strings.
`strcmp`	Compare strings.
`strmatch`	Find possible matches for a string.

You can also use the following `set` functions with cell arrays of strings.

Function	Description
`intersect`	Set the intersection of two vectors.
`ismember`	Detect members of a set.
`setdiff`	Return the set difference of two vectors.
`setxor`	Set the exclusive OR of two vectors.

Function	Description
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Formatting Strings

The following MATLAB functions offer the capability to compose a string that includes ordinary text and data formatted to your specification:

- `sprintf` — Write formatted data to an output string
- `fprintf` — Write formatted data to an output file or the command window
- `warning` — Display formatted data in a warning message
- `error` — Display formatted data in an error message and abort
- `assert` — Generate an error when a condition is violated

The syntax of each of these functions includes formatting operators similar to those used by the `printf` function in the C programming language. For example, `%s` tells MATLAB to interpret an input value as a string, and `%d` means to format an integer using decimal notation.

The general formatting syntax for these functions is

```
functionname(..., format_string, value1, value2, ..., valueN)
```

where the `format_string` argument expresses the basic content and formatting of the final output string, and the `value` arguments that follow supply data values to be inserted into the string.

Here is a sample `sprintf` statement, also showing the resulting output string:

```
sprintf('The price of %s on %d/%d/%d was $%.2f.', ...
        'bread', 7, 1, 2006, 2.49)
ans =
    The price of bread on 7/1/2006 was $2.49.
```

The following sections cover

- "The Format String" on page 2-43
- "Input Value Arguments" on page 2-44
- "The Formatting Operator" on page 2-45
- "Constructing the Formatting Operator" on page 2-46
- "Setting Field Width and Precision" on page 2-51
- "Restrictions for Using Identifiers" on page 2-54

Note The examples in this section of the documentation use only the `sprintf` function to demonstrate how string formatting works. However, you can run the examples using the `fprintf`, `warning`, and `error` functions as well.

The Format String

The first input argument in the `sprintf` statement shown above is the format string:

```
'The price of %s on %d/%d/%d was $%.2f.'
```

The `string` argument can include ordinary text, formatting operators and, in some cases, special characters. The formatting operators for this particular string are: `%s`, `%d`, `%d`, `%d`, and `%.2f`.

Following the string argument are five additional input arguments, one for each of the formatting operators in the string:

```
'bread', 7, 1, 2006, 2.49
```

When MATLAB processes the format string, it replaces each operator with one of these input values.

Special Characters. Special characters are a part of the text in the string. But, because they cannot be entered as ordinary text, they require a unique character sequence to represent them. Use any of the following character sequences to insert special characters into the output string.

To Insert . . .	Use . . .
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Backslash	`\\`
Percent character	`%%`

Input Value Arguments

In the syntax

```
functionname(..., format_string, value1, value2, ..., valueN)
```

The `value` arguments must immediately follow `string` in the argument list. In most instances, you supply one of these `value` arguments for each formatting operator used in `string`. Scalars, vectors, and numeric and character arrays are valid value arguments. You cannot use cell arrays or structures.

If you include fewer formatting operators than there are values to insert, MATLAB reuses the operators on the additional values. This example shows two formatting operators and six values to insert into the string:

```
sprintf('%s = %d\n', 'A', 479, 'B', 352, 'C', 651)
ans =
    A = 479
    B = 352
    C = 651
```

Sequential and Numbered Argument Specification.

You can place `value` arguments in the argument list either sequentially (that is, in the same order in which their formatting operators appear in the string), or by identifier (adding a number to each operator that identifies which `value` argument to replace it with). By default, MATLAB uses sequential ordering.

To specify arguments by a numeric identifier, add a positive integer followed by a $ sign immediately after the % sign in the operator. Numbered argument specification is explained in more detail under the topic "Value Identifiers" on page 2-51.

Ordered Sequentially	Ordered By Identifier

`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`     3rd 2nd 1st`

Vectorizing. Instead of using individual value arguments, you can use a vector or matrix as the source of data input values, as shown here:

```
sprintf('%d ', magic(4))
ans =
    16 5 9 4 2 11 7 14 3 10 6 15 13 8 12 1
```

When using the %s operator, MATLAB interprets integers as characters:

```
mvec = [77 65 84 76 65 66];

sprintf('%s ', mvec)
ans =
    MATLAB
```

The Formatting Operator

Formatting operators tell MATLAB how to format the numeric or character value arguments and where to insert them into the string. These operators control the notation, alignment, significant digits, field width, and other aspects of the output string.

A formatting operator begins with a % character, which may be followed by a series of one or more numbers, characters, or symbols, each playing a role in further defining the format of the insertion value. The final entry in this series is a single *conversion character* that MATLAB uses to define the notation style for the inserted data. Conversion characters used in MATLAB are based on those used by the `printf` function in the C programming language.

Here is a simple example showing five formatting variations for a common value:

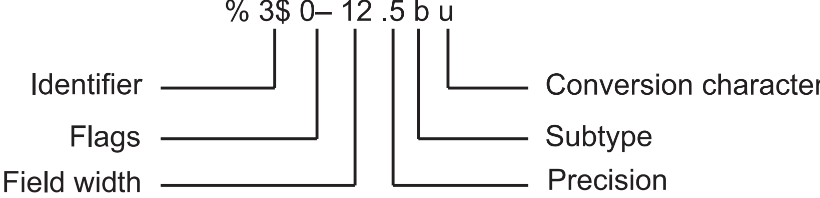
```
A = pi*100*ones(1,5);

sprintf(' %f \n %.2f \n %+.2f \n %12.2f \n %012.2f \n', A)
ans =
    314.159265       % Display in fixed-point notation (%f)
    314.16           % Display 2 decimal digits (%.2f)
    +314.16          % Display + for positive numbers (%+.2f)
          314.16     % Set width to 12 characters (%12.2f)
    000000314.16     % Replace leading spaces with 0 (%012.2f)
```

Constructing the Formatting Operator

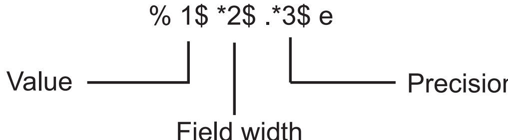
The fields that make up a formatting operator in MATLAB are as shown here, in the order they appear from right to left in the operator. The rightmost field, the conversion character, is required; the five to the left of that are optional. Each of these fields is explained in a section below:

- Conversion Character — Specifies the notation of the output.
- Subtype — Further specifies any nonstandard types.
- Precision — Sets the number of digits to display to the right of the decimal point.
- Field Width — Sets the minimum number of digits to display.
- Flags — Controls the alignment, padding, and inclusion of plus or minus signs.
- Value Identifiers — Map formatting operators to value input arguments. Use the identifier field when value arguments are not in a sequential order in the argument list.

Here is an example of a formatting operator that uses all six fields. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure).



An alternate syntax, that enables you to supply values for the field width and precision fields from values in the argument list, is shown below. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for information on when and how to use this syntax. (Again, space characters are shown only to improve readability of the figure.)



Each field of the formatting operator is described in the following sections. These fields are covered as they appear going from right to left in the formatting operator, starting with the `Conversion Character` and ending with the `Identifier` field.

Conversion Character. The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier. It is the only required field of the format specifier other than the leading `%` character.

Specifier	Description
`c`	Single character
`d`	Decimal notation (signed)
`e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`f`	Fixed-point notation

Specifier	Description
`g`	The more compact of `%e` or `%f`. (Insignificant zeros do not print.)
`G`	Same as `%g`, but using an uppercase `E`
`o`	Octal notation (unsigned)
`s`	String of characters
`u`	Decimal notation (unsigned)
`x`	Hexadecimal notation (using lowercase letters `a–f`)
`X`	Hexadecimal notation (using uppercase letters `A–F`)

This example uses conversion characters to display the number 46 in decimal, fixed-point, exponential, and hexadecimal formats:

```
A = 46*ones(1,4);

sprintf('%d   %f   %e   %X', A)
ans =
    46   46.000000   4.600000e+001   2E
```

Subtype. The subtype field is a single alphabetic character that immediately precedes the conversion character. The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

`b`	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like `'%bx'`.
`t`	The underlying C data type is a float rather than an unsigned integer.

Precision. `precision` in a formatting operator is a nonnegative integer that tells MATLAB how many digits to the right of the decimal point to use when formatting the corresponding input value. The `precision` field consists of a nonnegative integer that immediately follows a period and extends to the first alphabetic character after that period. For example, the specifier `%7.3f`, has a precision of 3.

Here are some examples of how the precision field affects different types of notation:

```
sprintf('%g   %.2g   %f   %.2f', pi*50*ones(1,4))
ans =
    157.08   1.6e+002   157.079633   157.08
```

Precision is not usually used in format specifiers for strings (i.e., %s). If you do use it on a string and if the value p in the precision field is less than the number of characters in the string, MATLAB displays only p characters of the string and truncates the rest.

You can also supply the value for a precision field from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for more information on this.

For more information on the use of precision in formatting, see "Setting Field Width and Precision" on page 2-51.

Field Width. Field width in a formatting operator is a nonnegative integer that tells MATLAB the minimum number of digits or characters to use when formatting the corresponding input value. For example, the specifier %7.3f, has a width of 7.

Here are some examples of how the width field affects different types of notation:

```
sprintf('|%e|%15e|%f|%15f|', pi*50*ones(1,4))
ans =
    |1.570796e+002|  1.570796e+002|157.079633|     157.079633|
```

When used on a string, the field width can determine whether MATLAB pads the string with spaces. If width is less than or equal to the number of characters in the string, it has no effect.

```
sprintf('%30s', 'Pad left with spaces')
ans =
          Pad left with spaces
```

You can also supply a value for `field width` from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for more information on this.

For more information on the use of `field width` in formatting, see "Setting Field Width and Precision" on page 2-51.

Flags. You can control the alignment of the output using any of these optional flags:

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field	`%-5.2d`
A plus sign (+)	Always prints a sign character (+ or –)	`%+5.2d`
Zero (0)	Pad with zeros rather than spaces.	`%05.2f`

Right- and left-justify the output. The default is to right-justify:

```
sprintf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
        12.3, 12.3)
ans =
    right-justify:       12.30
    left-justify: 12.30
```

Display a + sign for positive numbers. The default is to omit the + sign:

```
sprintf('no sign: %12.2f\nsign: %+12.2f', ...
        12.3, 12.3)
ans =
    no sign:       12.30
    sign:       +12.30
```

Pad to the left with spaces or zeros. The default is to use space-padding:

```
sprintf('space-padded: %12.2f\nzero-padded: %012.2f', ...
```

```
        5.2, 5.2)
ans =
    space-padded:         5.20
    zero-padded: 000000005.20
```

Note You can specify more than one flag in a formatting operator.

Value Identifiers. By default, MATLAB inserts data values from the argument list into the string in a sequential order. If you have a need to use the value arguments in a nonsequential order, you can override the default by using a numeric identifier in each format specifier. Specify nonsequential arguments with an integer immediately following the % sign, followed by a $ sign.

Ordered Sequentially	Ordered By Identifier

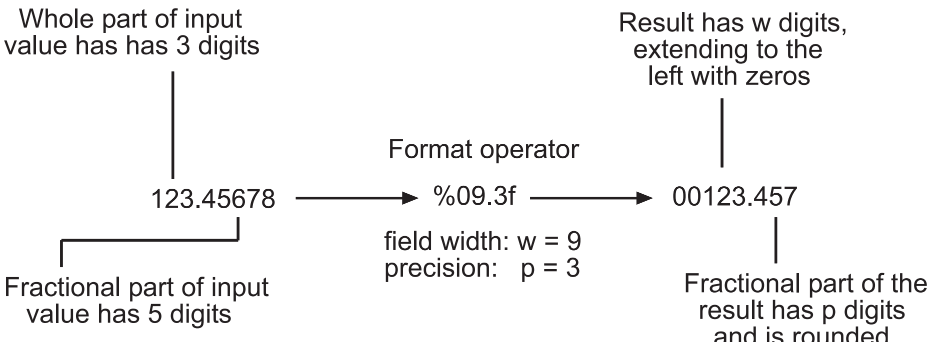
`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`     3rd 2nd 1st`

Setting Field Width and Precision

This section provides further information on the use of the field width and precision fields of the formatting operator:

- "Effect on the Output String" on page 2-51
- "Specifying Field Width and Precision Outside the format String" on page 2-52
- "Using Identifiers In the Width and Precision Fields" on page 2-53

Effect on the Output String. The figure below illustrates the way in which the field width and precision settings affect the output of the string formatting functions. In this figure, the zero following the % sign in the formatting operator means to add leading zeros to the output string rather than space characters:



General rules for formatting

- If precision is not specified, it defaults to 6.
- If precision (`p`) is less than the number of digits in the fractional part of the input value (`f`), then only `p` digits are shown to the right of the decimal point in the output, and that fractional value is rounded.
- If precision (`p`) is greater than the number of digits in the fractional part of the input value (`f`), then `p` digits are shown to the right of the decimal point in the output, and the fractional part is extended to the right with `p-f` zeros.
- If field width is not specified, it defaults to `precision` + 1 + the number of digits in the whole part of the input value.
- If field width (`w`) is greater than `p+1` plus the number of digits in the whole part of the input value (`n`), then the whole part of the output value is extended to the left with `w-(n+1+p)` space characters or zeros, depending on whether or not the `zero` flag is set in the `Flags` field. The default is to extend the whole part of the output with space characters.

Specifying Field Width and Precision Outside the format String. To specify field width or precision using values from a sequential argument list, use an asterisk (`*`) in place of the `field width` or `precision` field of the formatting operator.

This example formats and displays three numbers. The formatting operator for the first, `%*f`, has an asterisk in the field width location of the formatting

operator, specifying that just the field width, 15, is to be taken from the argument list. The second operator, `%.*f` puts the asterisk after the decimal point meaning, that it is the precision that is to take its value from the argument list. And the third operator, `%*.*f`, specifies both field width and precision in the argument list:

```
 sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...     % Width for 123.45678 is 15
        3, 16.42837, ...       % Precision for rand*20 is .3
        6, 4, pi)              % Width & Precision for pi is 6.4
ans =
     123.456780   16.428   3.1416
```

You can mix the two styles. For example, this statement gets the field width from the argument list and the precision from the format string:

```
sprintf('%*.2f', 5, 123.45678)
ans =
    123.46
```

Using Identifiers In the Width and Precision Fields. You can also derive field width and precision values from a nonsequential (i.e., numbered) argument list. Inside the formatting operator, specify `field width` and/or `precision` with an asterisk followed by an identifier number, followed by a $ sign.

This example from the previous section shows how to obtain field width and precision from a sequential argument list:

```
sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...
        3, 16.42837, ...
        6, 4, pi)

ans =
     123.456780   16.428   3.1416
```

Here is an example of how to do the same thing using numbered ordering. Field width for the first output value is 15, precision for the second value is 3, and field width and precision for the third value is 6 and 4, respectively.

If you specify field width or precision with identifiers, then you must specify the value with an identifier as well:

```
sprintf('%1$*4$f   %2$.*5$f   %3$*6$.*7$f', ...
123.45678, 16.42837, pi, 15, 3, 6, 4)

ans =
     123.456780   16.428   3.1416
```

Restrictions for Using Identifiers

If any of the formatting operators in a string include an identifier field, then all of the operators in that string must do the same; you cannot use both sequential and nonsequential ordering in the same function call.

Valid Syntax	Invalid Syntax

`sprintf('%d %d %d %d', ...`
`        1, 2, 3, 4)`
`ans =`
`    1 2 3 4`
	`sprintf('%d %3$d %d %d', ...`
	`        1, 2, 3, 4)`
	`ans =`
	`    1`

If your command provides more value arguments than there are formatting operators in the format string, MATLAB reuses the operators. However, MATLAB allows this only for commands that use sequential ordering. You cannot reuse formatting operators when making a function call with numbered ordering of the value arguments.

Valid Syntax	Invalid Syntax

`sprintf('%d', 1, 2, 3, 4)`
`ans =`
`    1234`
	`sprintf('%1$d', 1, 2, 3, 4)`
	`ans =`
	`    1`

Also, do not use identifiers when the value arguments are in the form of a vector or array:

Valid Syntax	Invalid Syntax

`v = [1.4 2.7 3.1];`
 	
`sprintf('%.4f %.4f %.4f', v)`
`ans =`
`    1.4000 2.7000 3.1000`
	`v = [1.4 2.7 3.1];`
	
	`sprintf('%3$.4f %1$.4f %2$.4f', v)`
	`ans =`
	`   Empty string: 1-by-0`

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.
- You can compare individual characters in two strings for equality.
- You can categorize every element within a string, determining whether each element is a character or white space.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings for Equality

You can use any of four functions to determine if two input strings are identical:

- `strcmp` determines if two strings are identical.
- `strncmp` determines if the first `n` characters of two strings are identical.
- `strcmpi` and `strncmpi` are the same as `strcmp` and `strncmp`, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns logical `0` (`false`). For example,

```
C = strcmp(str1,str2)
```

```
C =
    0
```

Note For C programmers, this is an important difference between the MATLAB `strcmp` and C `strcmp()`functions, where the latter returns 0 if the two strings are the same.

The first three characters of `str1` and `str2` are identical, so invoking `strncmp` with any value up to 3 returns `1`:

```
C = strncmp(str1, str2, 2)
C =
    1
```

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings

```
A = {'pizza'; 'chips'; 'candy'};
B = {'pizza'; 'chocolate'; 'pretzels'};
```

Now apply the string comparison functions:

```
strcmp(A,B)
ans =
    1
    0
    0
strncmp(A,B,1)
ans =
    1
    1
    0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine where the matching characters are in two strings:

```
A = 'fate';
B = 'cake';

A == B
ans =
     0    1    0    1
```

All of the relational operators (>, >=, <, <=, ==, ~=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are three functions for categorizing characters inside a string:

1. `isletter` determines if a character is a letter.
2. `isspace` determines if a character is white space (blank, tab, or new line).
3. `isstrprop` checks characters in a string to see if they match a category you specify, such as
   - Alphabetic
   - Alphanumeric
   - Lowercase or uppercase
   - Decimal digits
   - Hexadecimal digits
   - Control characters
   - Graphic characters
   - Punctuation characters
   - White-space characters

For example, create a string named `mystring`:

```
mystring = 'Room 401';
```

`isletter` examines each character in the string, producing an output vector of the same length as `mystring`:

```
A = isletter(mystring)
A =
     1    1    1    1    0    0    0    0
```

The first four elements in A are logical 1 (true) because the first four characters of mystring are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. (MATLAB also supports search and replace operations using regular expressions. See “Regular Expressions” on page 3-30.)

Consider a string named label:

```
label = 'Sample 1, 10/28/95';
```

The strrep function performs the standard search-and-replace operation. Use strrep to change the date from '10/28' to '10/30':

```
newlabel = strrep(label, '28', '30')
newlabel =
   Sample 1, 10/30/95
```

findstr returns the starting position of a substring within a longer string. To find all occurrences of the string 'amp' inside label, use

```
position = findstr('amp', label)
position =
     2
```

The position within label where the only occurrence of 'amp' begins is the second character.

The strtok function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of white-space characters. You can use the strtok function to parse a sentence into words. For example,

```
function allWords = words(inputString)
remainder = inputString;
allWords = '';
```

```
while (any(remainder))
  [chopped,remainder] = strtok(remainder);
  allWords = strvcat(allWords, chopped);
end
```

The `strmatch` function looks through the rows of a character array or cell array of strings to find strings that begin with a given series of characters. It returns the indices of the rows that begin with these characters:

```
maxstrings = strvcat('max', 'minimax', 'maximum')
maxstrings =
   max
   minimax
   maximum

strmatch('max', maxstrings)
ans =
     1
     3
```

Converting from Numeric to String

The functions listed in this table provide a number of ways to convert numeric data to character strings.

Function	Description	Example
`char`	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	`[72 105]` → `'Hi'`
`int2str`	Convert a positive or negative integer to a character type. (Rounds any fractional parts.)	`[72 105]` → `'72 105'`
`num2str`	Convert a numeric type to a character type of the specified precision and format.	`[72 105]` → `'72/105/'` (format set to `%1d/`)
`mat2str`	Convert a numeric type to a character type of the specified precision, returning a string MATLAB can evaluate.	`[72 105]` → `'[72 105]'`

Function	Description	Example
dec2hex	Convert a positive integer to a character type of hexadecimal base.	[72 105] → '48 69'
dec2bin	Convert a positive integer to a character type of binary base.	[72 105] → '1001000 1101001'
dec2base	Convert a positive integer to a character type of any base from 2 through 36.	[72 105] → '110 151' (base set to 8)

Converting to a Character Equivalent

The char function converts integers to Unicode character codes and returns a string composed of the equivalent characters:

```
x = [77 65 84 76 65 66];
char(x)
ans =
   MATLAB
```

Converting to a String of Numbers

The int2str, num2str, and mat2str functions convert numeric values to strings where each character represents a separate digit of the input value. The int2str and num2str functions are often useful for labeling plots. For example, the following lines use num2str to prepare automated labels for the *x*-axis of a plot:

```
function plotlabel(x, y)
plot(x, y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Converting to a Specific Radix

Another class of conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. This includes dec2hex, dec2bin, and dec2base.

Converting from String to Numeric

The functions listed in this table provide a number of ways to convert character strings to numeric data.

Function	Description	Example
`uintN` (e.g., `uint8`)	Convert a character to an integer code that represents that character.	`'Hi'` → `72 105`
`str2num`	Convert a character type to a numeric type.	`'72 105'` → `[72 105]`
`str2double`	Similar to `str2num`, but offers better performance and works with cell arrays of strings.	`{'72' '105'}` → `[72 105]`
`hex2num`	Convert a numeric type to a character type of specified precision, returning a string that MATLAB can evaluate.	`'A'` → `'-1.4917e-154'`
`hex2dec`	Convert a character type of hexadecimal base to a positive integer.	`'A'` → `10`
`bin2dec`	Convert a positive integer to a character type of binary base.	`'1010'` → `10`
`base2dec`	Convert a positive integer to a character type of any base from 2 through 36.	`'12'` → `10` (if `base == 8`)

Converting from a Character Equivalent

Character arrays store each character as a 16-bit numeric value. Use one of the integer conversion functions (e.g., `uint8`) or the `double` function to convert strings to their numeric values, and `char` to revert to character representation:

```
name = 'Thomas R. Lee';

name = double(name)
name =
    84   104   111   109   97   115   32   82   46   32   76   101   101

name = char(name)
name =
   Thomas R. Lee
```

Converting from a Numeric String

Use `str2num` to convert a character array to the numeric value represented by that string:

```
str = '37.294e-1';

val = str2num(str)
val =
    3.7294
```

The `str2double` function converts a cell array of strings to the double-precision values represented by the strings:

```
c = {'37.294e-1'; '-58.375'; '13.796'};

d = str2double(c)
d =
    3.7294
  -58.3750
   13.7960

whos
  Name      Size                       Bytes  Class

  c         3x1                          224  cell
  d         3x1                           24  double
```

Converting from a Specific Radix

To convert from a character representation of a nondecimal number to the value of that number, use one of these functions: `hex2num`, `hex2dec`, `bin2dec`, or `base2dec`.

The `hex2num` and `hex2dec` functions both take hexadecimal (base 16) inputs, but `hex2num` returns the IEEE double-precision floating-point number it represents, while `hex2dec` converts to a decimal integer.

Function Summary

MATLAB provides these functions for working with character arrays:

- Functions to Create Character Arrays on page 2-63
- Functions to Modify Character Arrays on page 2-63
- Functions to Read and Operate on Character Arrays on page 2-64
- Functions to Search or Compare Character Arrays on page 2-64
- Functions to Determine Data Type or Content on page 2-64
- Functions to Convert Between Numeric and String Data Types on page 2-65
- Functions to Work with Cell Arrays of Strings as Sets on page 2-65

Functions to Create Character Arrays

Function	Description
`'str'`	Create the string specified between quotes.
`blanks`	Create a string of blanks.
`sprintf`	Write formatted data to a string.
`strcat`	Concatenate strings.
`strvcat`	Concatenate strings vertically.

Functions to Modify Character Arrays

Function	Description
`deblank`	Remove trailing blanks.
`lower`	Make all letters lowercase.
`sort`	Sort elements in ascending or descending order.
`strjust`	Justify a string.
`strrep`	Replace one string with another.

Functions to Modify Character Arrays (Continued)

Function	Description
strtrim	Remove leading and trailing white space.
upper	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
eval	Execute a string with MATLAB expression.
sscanf	Read a string under format control.

Functions to Search or Compare Character Arrays

Function	Description
findstr	Find one string within another.
strcmp	Compare strings.
strcmpi	Compare strings, ignoring case.
strmatch	Find matches for a string.
strncmp	Compare the first N characters of strings.
strncmpi	Compare the first N characters, ignoring case.
strtok	Find a token in a string.

Functions to Determine Data Type or Content

Function	Description
iscellstr	Return true for a cell array of strings.
ischar	Return true for a character array.
isletter	Return true for letters of the alphabet.

Functions to Determine Data Type or Content (Continued)

Function	Description
isstrprop	Determine if a string is of the specified category.
isspace	Return true for white-space characters.

Functions to Convert Between Numeric and String Data Types

Function	Description
char	Convert to a character or string.
cellstr	Convert a character array to a cell array of strings.
double	Convert a string to numeric codes.
int2str	Convert an integer to a string.
mat2str	Convert a matrix to a string you can run eval on.
num2str	Convert a number to a string.
str2num	Convert a string to a number.
str2double	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets

Function	Description
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Dates and Times

In this section...

"Overview" on page 2-66

"Types of Date Formats" on page 2-66

"Conversions Between Date Formats" on page 2-68

"Date String Formats" on page 2-69

"Output Formats" on page 2-70

"Current Date and Time" on page 2-71

"Function Summary" on page 2-72

Overview

MATLAB represents date and time information in either of three formats: date strings, serial date numbers, or date vectors. You have the choice of using any of these formats. If you work with more than one date and time format, MATLAB provides functions to help you easily convert from one format to another, (e.g., from a string to a serial date number).

When using date strings, you have an additional option of choosing from 19 different string styles to express date and/or time information.

Types of Date Formats

The three MATLAB date and time formats are

- "Date Strings" on page 2-67
- "Serial Date Numbers" on page 2-67
- "Date Vectors" on page 2-68

This table shows examples of the three formats.

Date Format	Example
Date string	`02-Oct-1996`
Serial date number	`729300`
Date vector	`1996 10 2 0 0 0`

Date Strings

There are a number of different styles in which to express date and time information as a date string. For example, several possibilities for October 31, 2003 at 3:45:17 in the afternoon are

```
31-Oct-2003 15:45:17
10/31/03
15:45:17
03:45:17 PM
```

If you are working with a small number of dates at the MATLAB command line, then date strings are often the most convenient format to use.

Note The MATLAB `date` function returns the current date as a string.

Serial Date Numbers

A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '31-Oct-2003, 6:00 pm' in MATLAB is date number 731885.75.

MATLAB works internally with serial date numbers. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you get better performance if you use date numbers.

Note The MATLAB `now` function returns the current date and time as a serial date number.

Date Vectors

Date vectors are an internal format for some MATLAB functions; you do not typically use them in calculations. A date vector contains the elements `[year month day hour minute second]`.

Note The MATLAB `clock` function returns the current date and time as a serial vector.

Conversions Between Date Formats

Functions that convert between date formats are shown below.

Function	Description
`datenum`	Convert a date string to a serial date number.
`datestr`	Convert a serial date number to a date string.
`datevec`	Split a date number or date string into individual date elements.

Here are some examples of conversions from one date format to another:

```
d1 = datenum('02-Oct-1996')
d1 =
    729300

d2 = datestr(d1 + 10)
d2 =
   12-Oct-1996

dv1 = datevec(d1)
dv1 =
    1996        10         2         0         0         0

dv2 = datevec(d2)
dv2 =
    1996        10        12         0         0         0
```

Date String Formats

The datenum function is important for doing date calculations efficiently. datenum takes an input string in any of several formats, with 'dd-mmm-yyyy', 'mm/dd/yyyy', or 'dd-mmm-yyyy, hh:mm:ss.ss' most common. You can form up to six fields from letters and digits separated by any other characters:

- The day field is an integer from 1 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic string with at least three characters.
- The year field is a nonnegative integer: if only two digits are specified, then a year 19yy is assumed; if the year is omitted, then the current year is used as a default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by 'AM' or 'PM'.

For example, if the current year is 1996, then these are all equivalent:

```
'17-May-1996'
'17-May-96'
'17-May'
'May 17, 1996'
'5/17/96'
'5/17'
```

and both of these represent the same time:

```
'17-May-1996, 18:30'
'5/17/96/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus 3/6 is March 6, not June 3.

If you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros.

Output Formats

The command `datestr(D, dateform)` converts a serial date `D` to one of 19 different date string output formats showing date, time, or both. The default output for dates is a day-month-year string: 01-Mar-1996. You select an alternative output format by using the optional integer argument `dateform`.

This table shows the date string formats that correspond to each `dateform` value.

dateform	Format	Description
0	01-Mar-1996 15:45:17	day-month-year hour:minute:second
1	01-Mar-1996	day-month-year
2	03/01/96	month/day/year
3	Mar	month, three letters
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1996	year, four digits
11	96	year, two digits
12	Mar96	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	Q1-96	calendar quarter-year
18	Q1	calendar quarter

Converting Output Format with datestr

Here are some examples of converting the date March 1, 1996 to various forms using the `datestr` function:

```
d = '01-Mar-1999'
d =
   01-Mar-1999

datestr(d)
ans =
     01-Mar-1999

datestr(d, 2)
ans =
     03/01/99

datestr(d, 17)
ans =
     Q1-99
```

Current Date and Time

The function `date` returns a string for today's date:

```
date
ans =
     02-Oct-1996
```

The function `now` returns the serial date number for the current date and time:

```
now
ans =
     729300.71

datestr(now)
ans =
     02-Oct-1996 16:56:16

datestr(floor(now))
ans =
     02-Oct-1996
```

Function Summary

MATLAB provides the following functions for time and date handling:

- Current Date and Time Functions on page 2-72
- Conversion Functions on page 2-72
- Utility Functions on page 2-72
- Timing Measurement Functions on page 2-73

Current Date and Time Functions

Function	Description
`clock`	Return the current date and time as a date vector.
`date`	Return the current date as date string.
`now`	Return the current date and time as serial date number.

Conversion Functions

Function	Description
`datenum`	Convert to a serial date number.
`datestr`	Convert to a string representation of the date.
`datevec`	Convert to a date vector.

Utility Functions

Function	Description
`addtodate`	Modify a date number by field.
`calendar`	Return a matrix representing a calendar.
`datetick`	Label axis tick lines with dates.

Utility Functions (Continued)

Function	Description
`eomday`	Return the last day of a year and month.
`weekday`	Return the current day of the week.

Timing Measurement Functions

Function	Description
`cputime`	Return the total CPU time used by MATLAB since it was started.
`etime`	Return the time elapsed between two date vectors.
`tic`, `toc`	Measure the time elapsed between invoking `tic` and `toc`.

Structures

In this section...

"Overview" on page 2-74

"Building Structure Arrays" on page 2-75

"Accessing Data in Structure Arrays" on page 2-78

"Using Dynamic Field Names" on page 2-80

"Finding the Size of Structure Arrays" on page 2-81

"Adding Fields to Structures" on page 2-82

"Deleting Fields from Structures" on page 2-83

"Applying Functions and Operators" on page 2-83

"Writing Functions to Operate on Structures" on page 2-84

"Organizing Data in Structure Arrays" on page 2-85

"Nesting Structures" on page 2-91

"Function Summary" on page 2-92

Overview

Structures are MATLAB arrays with named "data containers" called *fields*. The fields of a structure can contain any kind of data. For example, one field might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on.

```
patient
 |
 |-.name ----------- 'John Doe'
 |-.billing -------- 127.00
 |-.test ----------- 79   75   73
                     180  178  177.5
                     220  210  205
```

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional structure arrays. For examples of higher-dimension structure arrays, see "Multidimensional Arrays" on page 1-56.

Building Structure Arrays

You can build structures in two ways:

- Using assignment statements
- Using the `struct` function

Building Structure Arrays Using Assignment Statements

You can build a simple 1-by-1 structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the 1-by-1 `patient` structure array shown at the beginning of this section:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering

```
patient
```

at the command line results in

```
name: 'John Doe'
billing: 127
test: [3x3 double]
```

`patient` is an array containing a structure with three fields. To expand the structure array, add subscripts after the structure name:

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
```

The `patient` structure array now has size `[1 2]`. Note that once a structure array contains more than a single element, MATLAB does not display individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains:

```
patient
patient =
1x2 struct array with fields:
    name
    billing
    test
```

You can also use the `fieldnames` function to obtain this information. `fieldnames` returns a cell array of strings containing field names.

As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that

- All structures in the array have the same number of fields.
- All fields have the same field names.

For example, entering `patient(3).name = 'Alan Johnson'` expands the `patient` array to size `[1 3]`. Now both `patient(3).billing` and `patient(3).test` contain empty matrices.

Note Field sizes do not have to conform for every element in an array. In the `patient` example, the `name` fields can have different lengths, the `test` fields can be arrays of different sizes, and so on.

Building Structure Arrays Using the struct Function

You can preallocate an array of structures with the `struct` function. Its basic form is

```
strArray = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their corresponding values. A field value can be a single value, represented by any MATLAB data construct, or a cell array of values. All field values in the argument list must be of the same scale (single value or cell array).

You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an example, consider the allocation of a 1-by-3 structure array, `weather`, with the structure fields `temp` and `rainfall`. Three different methods for allocating such an array are shown in this table.

Method	Syntax	Initialization
`struct`	`weather(3) = struct('temp', 72, ... 'rainfall', 0.0);`	`weather(3)` is initialized with the field values shown. The fields for the other structures in the array, `weather(1)` and `weather(2)`, are initialized to the empty matrix.
`struct` with `repmat`	`weather = repmat(struct('temp', ... 72, 'rainfall', 0.0), 1, 3);`	All structures in the `weather` array are initialized using one set of field values.
`struct` with cell array syntax	`weather = ...` `struct('temp', {68, 80, 72}, ... 'rainfall', {0.2, 0.4, 0.0});`	The structures in the `weather` array are initialized with distinct field values specified with cell arrays.

Naming conventions for Structure Field Names

MATLAB structure field names are required to follow the same rules as standard MATLAB variables:

1. Field names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. The following statements are all invalid:

   ```
   w = setfield(w, 'My.Score', 3);
   ```

```
w = setfield(w, '1stScore', 3);
w = setfield(w, '1+1=3', 3);
w = setfield(w, '@MyScore', 3);
```

2 Although field names can be of any length, MATLAB uses only the first N characters of the field name, (where N is the number returned by the function `namelengthmax`), and ignores the rest.

```
N= namelengthmax
N=
   63
```

3 MATLAB distinguishes between uppercase and lowercase characters. Field name `length` is not the same as field name `Length`.

4 In most cases, you should refrain from using the names of functions and variables as field names.

See “Adding Fields to Structures” on page 2-82 and “Deleting Fields from Structures” on page 2-83 for more information on working with field names.

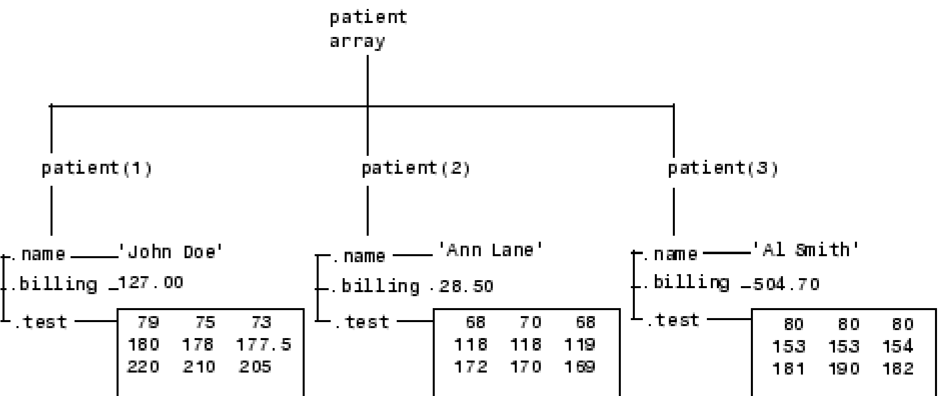
Memory Requirements for Structures

You do not necessarily need a contiguous block of memory to store a structure. The memory for each field in the structure needs to be contiguous, but not the entire structure itself.

Accessing Data in Structure Arrays

Using structure array indexing, you can access the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. You can also access the fields of an array of structures in the form of a comma-separated list.

For the examples in this section, consider this structure array.



You can access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-2 structure array:

```
mypatients = patient(1:2)
1x2 struct array with fields:
    name
    billing
    test
```

The first structure in the `mypatients` array is the same as the first structure in the `patient` array:

```
mypatients(1)
ans =
       name: 'John Doe'
    billing: 127
       test: [3x3 double]
```

To access a field of a particular structure, include a period (.) after the structure name followed by the field name:

```
str = patient(2).name
str =
   Ann Lane
```

To access elements within fields, append the appropriate indexing mechanism to the field name. That is, if the field contains an array, use array subscripting; if the field contains a cell array, use cell array subscripting, and so on:

```
test2b = patient(3).test(2,2)
test2b =
   153
```

Use the same notations to assign values to structure fields, for example,

```
patient(3).test(2,2) = 7;
```

You can extract field values for multiple structures at a time. For example, the line below creates a 1-by-3 vector containing all of the `billing` fields:

```
bills = [patient.billing]
bills =
  127.0000   28.5000  504.7000
```

Similarly, you can create a cell array containing the `test` data for the first two structures:

```
tests = {patient(1:2).test}
tests =
    [3x3 double]    [3x3 double]
```

Using Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate *`expression`* into a field name and obtain the values of that field at columns `1` through `25` of row `7`, use

```
structName.(expression)(7,1:25)
```

Dynamic Field Names Example

The avgscore function shown below computes an average test score, retrieving information from the testscores structure using dynamic field names:

```
function avg = avgscore(student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field student. First, initialize the structure that contains scores for a 25 week period:

```
testscores.Ann_Lane.week(1:25) = ...
  [95 89 76 82 79 92 94 92 89 81 75 93 ...
   85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ...
  [87 80 91 84 99 87 93 87 97 87 82 89 ...
   86 82 90 98 75 79 92 84 90 93 84 78 81];
```

Now run avgscore, supplying the students name fields for the testscores structure at runtime using dynamic field names:

```
avgscore(testscores, 'Ann_Lane', 7, 22)
ans =
   85.2500

avgscore(testscores, 'William_King', 7, 22)
ans =
   87.7500
```

Finding the Size of Structure Arrays

Use the size function to obtain the size of a structure array, or of any structure field. Given a structure array name as an argument, size returns a vector of array dimensions. Given an argument in the form array(n).field, the size function returns a vector containing the size of the field contents.

For example, for the 1-by-3 structure array patient, size(patient) returns the vector [1 3]. The statement size(patient(1,2).name) returns the length of the name string for element (1,2) of patient.

Adding Fields to Structures

You can add a field to every structure in an array by adding the field to a single structure. For example, to add a social security number field to the patient array, use an assignment like

```
patient(2).ssn = '000-00-0000';
```

Now patient(2).ssn has the assigned value. Every other structure in the array also has the ssn field, but these fields contain the empty matrix until you explicitly assign a value to them.

See “Naming conventions for Structure Field Names” on page 2-77 for guidelines to creating valid field names.

Adding or Modifying Fields With the setfield Function

The setfield function offers another way to add or modify fields of a structure. Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1;
```

You can change the name field of mystr(2,1) using

```
mystr = setfield(mystr, {2,1}, 'name', 'ted');

mystr(2,1).name
ans =
   ted
```

Adding New Fields Dynamically

To add new fields to a structure, specifying the names for these fields at run-time, see the section on “Using Dynamic Field Names” on page 2-80.

Deleting Fields from Structures

You can remove a given field from every structure within a structure array using the `rmfield` function. Its most basic form is

```
struc2 = rmfield(array, 'field')
```

where `array` is a structure array and `'field'` is the name of a field to remove from it. To remove the `name` field from the `patient` array, for example, enter

```
patient = rmfield(patient, 'name');
```

Applying Functions and Operators

Operate on fields and field elements the same way you operate on any other MATLAB array. Use indexing to access the data on which to operate.

For example, this statement finds the mean across the rows of the `test` array in `patient(2)`:

```
mean((patient(2).test)');
```

There are sometimes multiple ways to apply functions or operators across fields in a structure array. One way to add all the `billing` fields in the `patient` array is

```
total = 0;
for k = 1:length(patient)
      total = total + patient(k).billing;
end
```

To simplify operations like this, MATLAB enables you to operate on all like-named fields in a structure array. Simply enclose the `array.field` expression in square brackets within the function call. For example, you can sum all the `billing` fields in the `patient` array using

```
total = sum ([patient.billing]);
```

This is equivalent to using the comma-separated list:

```
total = sum ([patient(1).billing, patient(2).billing, ...]);
```

This syntax is most useful in cases where the operand field is a scalar field:

Writing Functions to Operate on Structures

You can write functions that work on structures with specific field architectures. Such functions can access structure fields and elements for processing.

Note When writing M-file functions to operate on structures, you must perform your own error checking. That is, you must ensure that the code checks for the expected fields.

As an example, consider a collection of data that describes measurements, at different times, of the levels of various toxins in a water source. The data consists of fifteen separate observations, where each observation contains three separate measurements.

You can organize this data into an array of 15 structures, where each structure has three fields, one for each of the three measurements taken.

The function `concen`, shown below, operates on an array of structures with specific characteristics. Its arguments must contain the fields `lead`, `mercury`, and `chromium`:

```
function [r1, r2] = concen(toxtest);
% Create two vectors:
%  r1 contains the ratio of mercury to lead at each observation.
%  r2 contains the ratio of lead to chromium.
r1 = [toxtest.mercury] ./ [toxtest.lead];
r2 = [toxtest.lead] ./ [toxtest.chromium];

% Plot the concentrations of lead, mercury, and chromium
% on the same plot, using different colors for each.
lead = [toxtest.lead];
mercury = [toxtest.mercury];
chromium = [toxtest.chromium];

plot(lead, 'r');   hold on
```

```
plot(mercury, 'b')
plot(chromium, 'y');   hold off
```

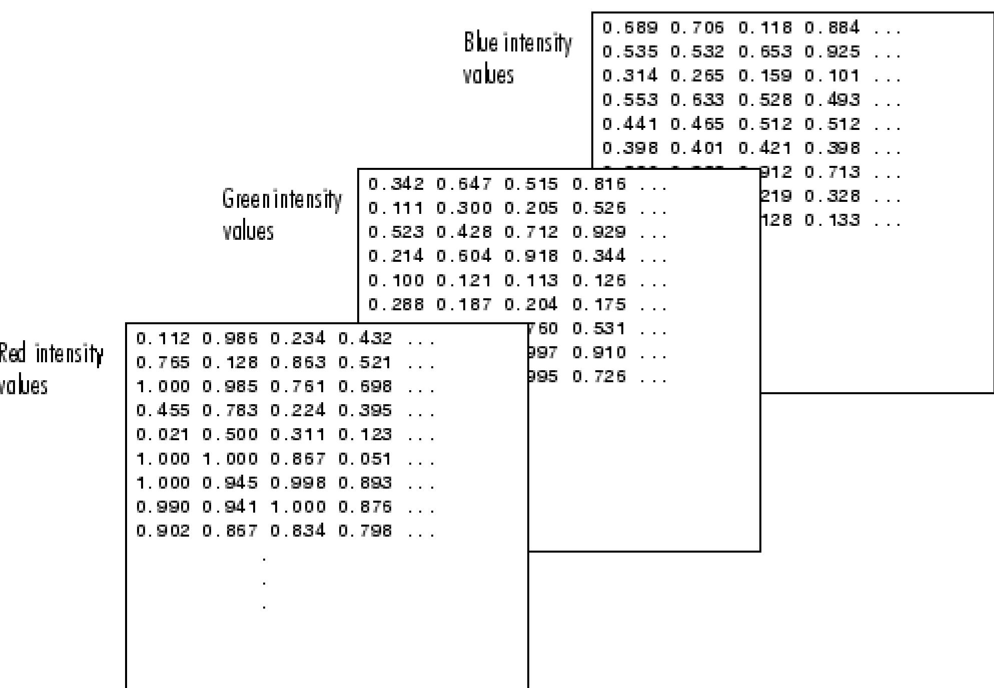
Try this function with a sample structure array like `test`:

```
test(1).lead = .007;
test(2).lead = .031;
test(3).lead = .019;

test(1).mercury = .0021;
test(2).mercury = .0009;
test(3).mercury = .0013;

test(1).chromium = .025;
test(2).chromium = .017;
test(3).chromium = .10;
```

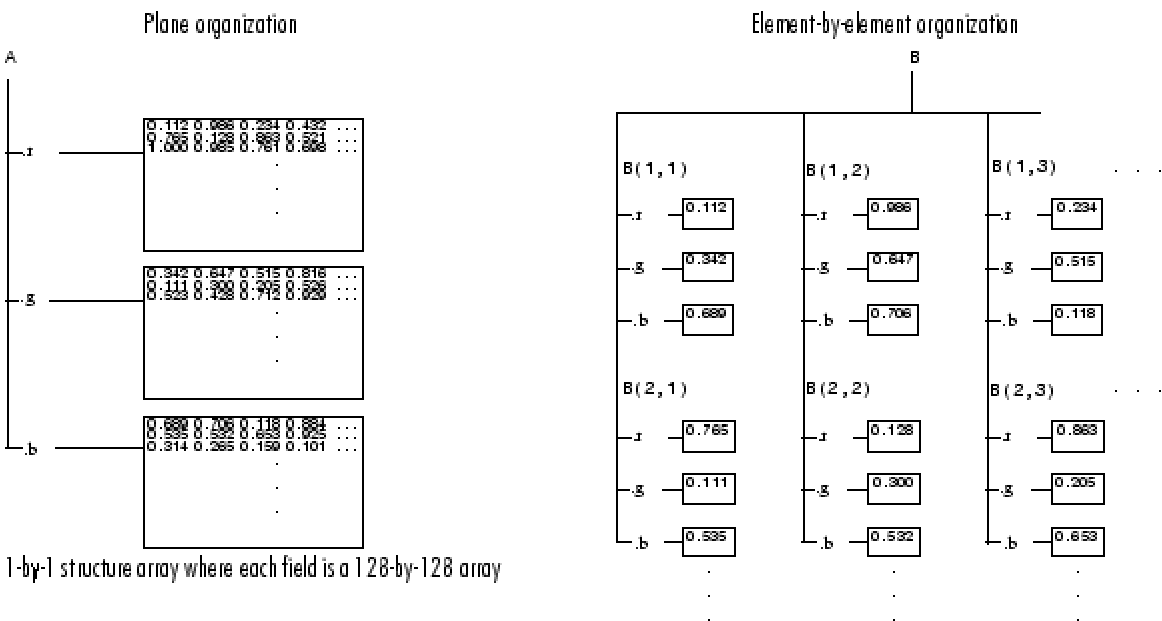
Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; `RED`, `GREEN`, and `BLUE`.


Blue intensity values
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
0.398 0.401 0.421 0.398 ...
0.713 ...
0.328 ...
0.133 ...
Green intensity values
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
0.288 0.187 0.204 0.175 ...
0.531 ...
0.910 ...
0.726 ...
Red intensity values
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
.
.
.

There are at least two ways you can organize such data into a structure array.



1-by-1 structure array where each field is a 128-by-128 array

128-by-128 structure array where each field is a single data element

Plane Organization

In the plane organization, shown to the left in the figure above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
redPlane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as `A(2)`, `A(3)`, and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately:

```
redSub = A.r(2:12,13:30);
greenSub = A.g(2:12,13:30);
blueSub = A.b(2:12,13:30);
```

Element-by-Element Organization

The element-by-element organization, shown to the right in the figure above, has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```
Bsub = B(1:10,1:10);
```

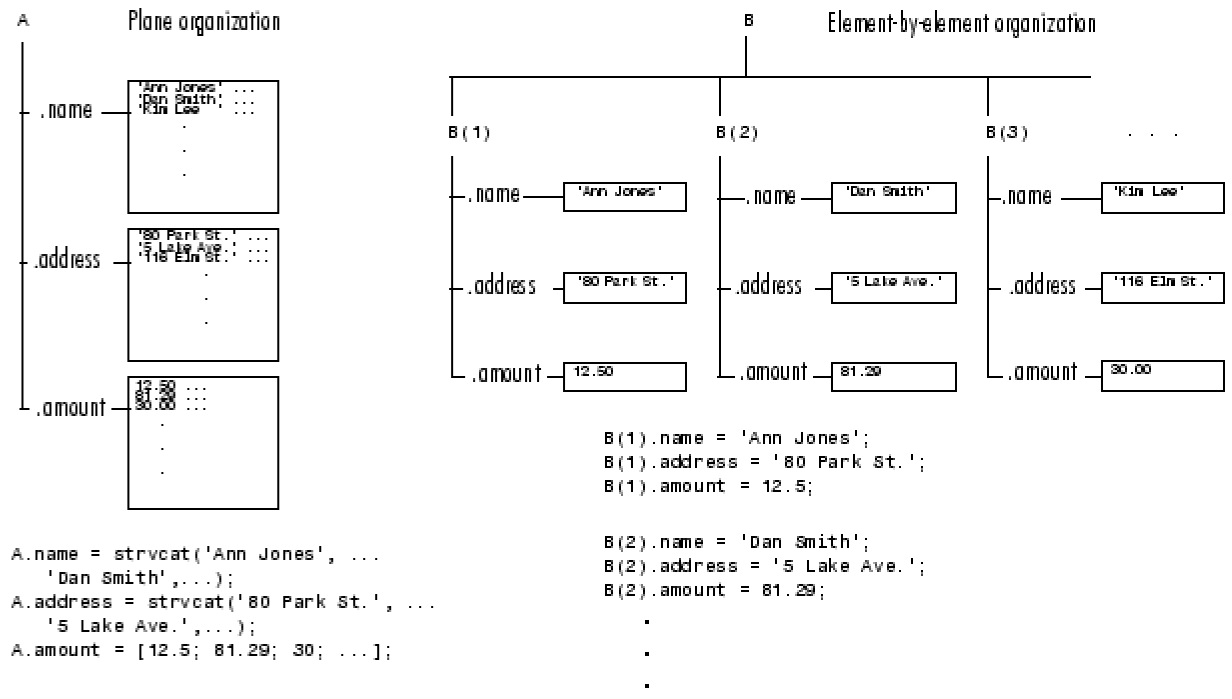
To access an entire plane of the image using the element-by-element method, however, requires a loop:

```
redPlane = zeros(128, 128);
for k = 1:(128 * 128)
    redPlane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications; however, it can be the best for other applications wherein you will routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example — A Simple Database

Consider organizing a simple database.



Each of the possible organizations has advantages depending on how you want to access the data:

- Plane organization makes it easier to operate on all field values at once. For example, to find the average of all the values in the `amount` field,
  - Using plane organization

```
avg = mean(A.amount);
```

- Using element-by-element organization

```
avg = mean([B.amount]);
```

- Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, `client.m`, which displays the name and address of a given client on screen.

  Using plane organization, pass individual fields.

  ```
  function client(name,address)
  disp(name)
  disp(address)
  ```

  To call the `client` function,

  ```
  client(A.name(2,:),A.address(2,:))
  ```

  Using element-by-element organization, pass an entire structure.

  ```
  function client(B)
  disp(B)
  ```

  To call the `client` function,

  ```
  client(B(2))
  ```

- Element-by-element organization makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you may need to frequently recreate the `name` or `address` field to accommodate longer strings.

Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Nesting Structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the `struct` function or direct assignment statements to nest structures within existing structure fields.

Building Nested Structures with the struct Function

To build nested structures, you can nest calls to the `struct` function. For example, create a 1-by-1 structure array:

```
A = struct('data', [3 4 7; 8 0 1], 'nest',...
    struct('testnum', 'Test 1', 'xdata', [4 2 8],...
    'ydata', [7 1 6]));
```

You can build nested structure arrays using direct assignment statements. These statements add a second element to the array:

```
A(2).data = [9 3 2; 7 6 5];
A(2).nest.testnum = 'Test 2';
A(2).nest.xdata = [3 4 2];
A(2).nest.ydata = [5 0 9];
```

Indexing Nested Structures

To index nested structures, append nested field names using dot notation. The first text string in the indexing expression identifies the structure array, and subsequent expressions access field names that contain other structures.

For example, the array `A` created earlier has three levels of nesting:

- To access the nested structure inside `A(1)`, use `A(1).nest`.
- To access the `xdata` field in the nested structure in `A(2)`, use `A(2).nest.xdata`.
- To access element 2 of the `ydata` field in `A(1)`, use `A(1).nest.ydata(2)`.

Function Summary

This table describes the MATLAB functions for working with structures.

Function	Description
`deal`	Deal inputs to outputs.
`fieldnames`	Get structure field names.
`isfield`	Return `true` if the field is in a structure array.
`isstruct`	Return `true` for structures.
`rmfield`	Remove a structure field.
`struct`	Create or convert to a structure array.
`struct2cell`	Convert a structure array into a cell array.

Cell Arrays

In this section...

"Overview" on page 2-93

"Cell Array Operators" on page 2-94

"Creating a Cell Array" on page 2-95

"Referencing Cells of a Cell Array" on page 2-99

"Deleting Cells" on page 2-106

"Reshaping Cell Arrays" on page 2-106

"Replacing Lists of Variables with Cell Arrays" on page 2-107

"Applying Functions and Operators" on page 2-108

"Organizing Data in Cell Arrays" on page 2-109

"Nesting Cell Arrays" on page 2-110

"Converting Between Cell and Numeric Arrays" on page 2-112

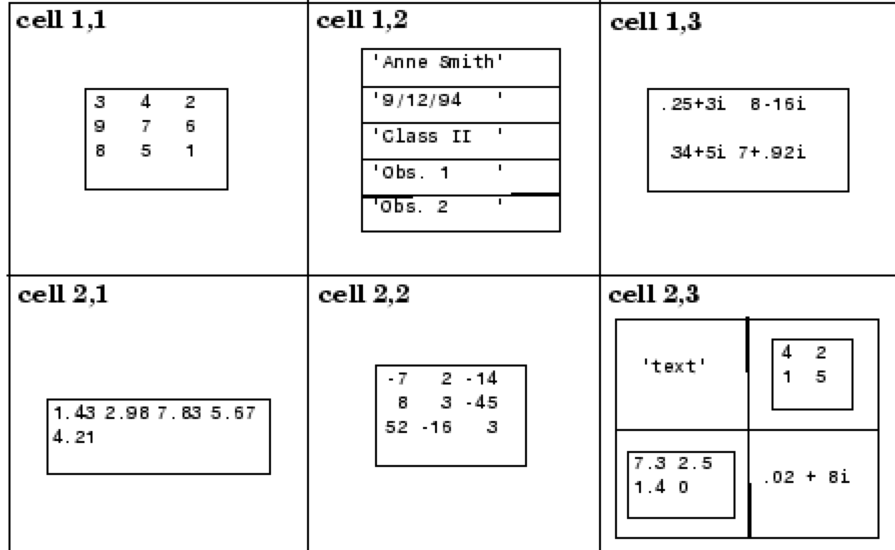
"Cell Arrays of Structures" on page 2-113

"Function Summary" on page 2-114

Overview

A cell array provides a storage mechanism for dissimilar kinds of data. You can store arrays of different types and/or sizes within the cells of a cell array. For example, you can store a 1-by-50 char array, a 7-by-13 double array, and a 1-by-1 `uint32` in cells of the same cell array.

This illustration shows a cell array `A` that contains arrays of unsigned integers in `A{1,1}`, strings in `A{1,2}`, complex numbers in `A{1,3}`, floating-point numbers in `A{2,1}`, signed integers in `A{2,2}`, and another cell array in `A{2,3}`.



To access data in a cell array, you use the same type of matrix indexing as with other MATLAB matrices and arrays. However, with cell array indexing, you use curly braces, `{}`, instead of square brackets or parentheses around the array indices. For example, `A{2,3}` accesses the cell in row 2 and column 3 of cell array `A`.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see “Multidimensional Arrays” on page 1-56.

Cell Array Operators

This table shows the operators used in constructing, concatenating, and indexing into the cells of a cell array.

Operation	Syntax	Description
Constructing	C = {A B D E}	Builds a cell array C that can contain data of unlike types in A, B, D, and E
Concatenating	C3 = {C1 C2}	Concatenates cell arrays C1 and C2 into a 2–element cell array C3 such that C3{1} = C1 and C3{2} = C2
	C3 = [C1 C2]	Concatenates the *contents* of cell arrays C1 and C2
Indexing	X = C(s)	Returns the *cells* of array C that are specified by subscripts s
	X = C{s}	Returns the *contents* of the cells of C that are specified by subscripts s
	X = C{s}(t)	References one or more elements of an array that resides within a cell. Subscript s selects the cell, and subscript t selects the array element(s).

Creating a Cell Array

Creating cell arrays in MATLAB is similar to creating arrays of other MATLAB data types like double, character, etc. The main difference is that, when constructing a cell array, you enclose the array contents or indices with curly braces { } instead of square brackets [ ]. The curly braces are cell array constructors, just as square brackets are numeric array constructors. Use commas or spaces to separate elements and semicolons to terminate each row.

For example, to create a 2-by-2 cell array A, type

```
A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

This results in the array shown below:

cell 1,1	cell 1,2
1 4 3 0 5 8 7 2 9	'Anne Smith'
cell 2,1	**cell 2,2**
3+7i	[-3.14...3.14]

Note The notation {} denotes the empty cell array, just as [] denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

For more information on cell arrays, refer to these topics:

- "Creating Cell Arrays Using Multiple Assignment Statements" on page 2-96
- "Building Cell Arrays with Concatenation" on page 2-98
- "Preallocating Cell Arrays with the cell Function" on page 2-99
- "Memory Requirements for Cell Arrays" on page 2-99

Creating Cell Arrays Using Multiple Assignment Statements

You also can create a cell array one cell at a time. MATLAB expands the size of the cell array with each assignment statement:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/4:pi};
```

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening cells with empty matrices. For example, the assignment below turns the 2-by-2 cell array A into a 3-by-3 cell array.

```
A(3,3) = {5};
```

cell 1,1	cell 1,2	cell 1,3
1 4 3 0 5 8 7 2 9	'Anne Smith'	[ ]
cell 2,1	**cell 2,2**	**cell 2,3**
3+7i	[-3.14...3.14]	[ ]
cell 3,1	**cell 3,2**	**cell 3,3**
[ ]	[ ]	5

3–by-3 Cell Array

Note If you already have a numeric array of a given name, don't try to create a cell array of the same name by assignment without first clearing the numeric array. If you do not clear the numeric array, MATLAB assumes that you are trying to "mix" cell and numeric syntaxes, and generates an error. Similarly, MATLAB does not clear a cell array when you make a single assignment to it. If any of the examples in this section give unexpected results, clear the cell array from the workspace and try again.

Alternative Assignment Syntax. When assigning values to a cell array, either of the syntaxes shown below is valid. You can use the braces on the right side of the equation, enclosing the value being assigned as shown here:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
```

Or use them on the left side, enclosing the array subscripts:

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
```

Building Cell Arrays with Concatenation

There are two ways that you can construct a new cell array from existing cell arrays:

- Concatenate entire cell arrays to individual cells of the new array. For example, join three cell arrays together to build a new cell array having three elements, each containing a cell array. This method uses the curly brace `{ }` operator.
- Concatenate the *contents* of the cells into a new array. For example, join cell arrays of size `m-by-n1`, `m-by-n2`, and `m-by-n3` together to yield a new cell array that is `m-by-(n1+n2+n3)` in size. This method uses the square bracket `[ ]` operator.

Here is an example. First, create three 3–row cell arrays of different widths.

```
C1 = {'Jan' 'Feb';  '10' '17';  uint16(2004) uint16(2001)};
C2 = {'Mar' 'Apr' 'May';  '31' '2' '10';  ...
      uint16(2006) uint16(2005) uint16(1994)};
C3 = {'Jun';  '23';  uint16(2002)};
```

This creates arrays C1, C2, and C3:

```
      C1                  C2                  C3
'Jan'   'Feb'       'Mar'   'Apr'   'May'       'Jun'
'10'    '17'        '31'    '2'     '10'        '23'
[2004]  [2001]      [2006]  [2005]  [1994]      [2002]
```

Use the curly brace operator to concatenate entire cell arrays, thus building a 1-by-3 cell array from the three initial arrays. Each cell of this new array holds its own cell array:

```
C4 = {C1 C2 C3}
C4 =
    {3x2 cell}    {3x3 cell}    {3x1 cell}
```

Now use the square bracket operator on the same combination of cell arrays. This time MATLAB concatenates the contents of the cells together and produces a 3-by-6 cell array:

```
C5 = [C1 C2 C3]
```

```
C5 =
    'Jan'     'Feb'     'Mar'     'Apr'     'May'     'Jun'
    '10'      '17'      '31'      '2'       '10'      '23'
    [2004]    [2001]    [2006]    [2005]    [1994]    [2002]
```

Preallocating Cell Arrays with the cell Function

The `cell` function enables you to preallocate empty cell arrays of the specified size. For example, this statement creates an empty 20-by-30 cell array:

```
B = cell(20, 30);
```

Use assignment statements to fill the cells of `B`.

It is more efficient to preallocate a cell array of a required size using the `cell` function and then assign data into it, than to grow a cell array as you go along using individual data assignments. The `cell` function, therefore, offers the most memory-efficient way of preallocating a cell array.

Memory Requirements for Cell Arrays

You do not necessarily need a contiguous block of memory to store a cell array. The memory for each cell needs to be contiguous, but not the entire array of cells.

Referencing Cells of a Cell Array

Because a cell array can contain different types of data stored in various array sizes, cell array indexing is a little more complex than indexing into a numeric or character array.

This section covers the following topics on constructing a cell array:

- "Manipulating Cells and the Contents of Cells" on page 2-100
- "Working With Arrays Within Cells" on page 2-103
- "Working With Structures Within Cells" on page 2-103
- "Working With Cell Arrays Within Cells" on page 2-104
- "Plotting the Cell Array" on page 2-105

The examples in this section illustrate how to access the different components of a cell array. All of the examples use the following six-cell array which consists of different data types.

First, build the individual components of the example array:

```
rand('state', 0);    numArray = rand(3,5)*20;
chArray = ['Ann Lane'; 'John Doe'; 'Al Smith'];
cellArray = {1 4 3 9; 0 5 8 2; 7 2 9 2; 3 3 1 4};
logArray = numArray > 10;

stArray(1).name = chArray(1,:);
stArray(2).name = chArray(2,:);
stArray(1).billing = 28.50;
stArray(2).billing = 139.72;
stArray(1).test = numArray(1,:);
stArray(2).test = numArray(2,:);
```

and then construct the cell array from these components using the `{ }` operator:

```
A = {numArray, pi, stArray; chArray, cellArray, logArray};
```

To see what size and type of array occupies each cell in `A`, type the variable name alone:

```
A
A =
    [3x5 double]    [  3.1416]    [1x2 struct ]
    [3x8 char  ]    {4x4 cell}    [3x5 logical]
```

Manipulating Cells and the Contents of Cells

When working with cell arrays, you have a choice of selecting entire cells of an array to work with, or the contents of those cells. The first method is called *cell indexing*; the second is *content indexing*:

- Cell indexing enables you to work with whole cells of an array. You can access single or multiple cells within the array, but you cannot select anything less than the complete cell. If you want to manipulate the cells

of an array without regard to the contents of those cells, use cell indexing. This type of indexing is denoted by the parentheses operator `( )`.

Use cell indexing to assign any set of cells to another variable, creating a new cell array.

Creating a New Cell Array from an Existing One

- Content indexing gives you access to the contents of a cell. You can work with individual elements of an array within a cell, but you can only do so for one cell at a time. This indexing uses the curly brace operator `{ }`.

Displaying Parts of the Cell Array. Using the example cell array `A`, you can display information on the first row of cells using cell indexing. (The MATLAB colon operator functions the same when used with cell arrays as it does with numeric arrays):

```
A(1,:)
ans =
    [3x5 double]    [3.1416]    [1x2 struct]
```

To display the contents of these cells, use content indexing:

```
A{1,:}
ans =
   19.0026    9.7196    9.1294    8.8941   18.4363
    4.6228   17.8260    0.3701   12.3086   14.7641
   12.1369   15.2419   16.4281   15.8387    3.5253
ans =
    3.1416
ans =
1x2 struct array with fields:
    name
```

```
billing
test
```

In assignments, you can use content indexing to access only a single cell, not a subset of cells. For example, the statements `A{1,:} = value` and `B = A{1,:}` are both invalid. However, you can use a subset of cells any place you would normally use a comma-separated list of variables (for example, as function inputs or when building an array). See "Replacing Lists of Variables with Cell Arrays" on page 2-107 for details.

Assigning Cells. For cell indexing, assign the `double` array cell to `X`:

```
X = A(1,1)
X =
    [3x5 double]
```

`X` is a 1-by-1 cell array:

```
whos X
  Name      Size                    Bytes  Class

  X         1x1                       180  cell
```

For content indexing, assign the contents of the first cell of row 1 to `Y`:

```
Y = A{1,1}
Y =
   19.0026    9.7196    9.1294    8.8941   18.4363
    4.6228   17.8260    0.3701   12.3086   14.7641
   12.1369   15.2419   16.4281   15.8387    3.5253
```

`Y` is a 3-by-5 `double` array

```
whos Y
  Name      Size                    Bytes  Class

  Y         3x5                       120  double
```

Assigning Multiple Cells. Assigning multiple cells with cell indexing is similar to assigning a single cell. MATLAB creates a new cell array, each cell of which contains a cell array.

Create a 1-by-2 array with cells from A(1,2) and A(1,3):

```
X = A(1,2:3)
X =
    [3.1416]    [1x2 struct]

whos X
  Name      Size                    Bytes  Class

  X         1x2                       808  cell
```

But assigning the contents of multiple cells returns a comma-separated list. In this case, you need one output variable on the left side of the assignment statement for each cell on the right side:

```
[Y1 Y2] = A{1,2:3}
Y1 =
    3.1416
Y2 =
1x2 struct array with fields:
    name
    billing
    test
```

Working With Arrays Within Cells

Append the parentheses operator to the cell designator A{1,1} to select specific elements of a cell. This example displays specific row and columns of the numeric array stored in cell {1,1} of A:

```
A{1,1}(2,3:end)
ans =
    0.3701   12.3086   14.7641
```

Working With Structures Within Cells

Use a combination of indexing operators to access the components of a structure array that resides in a cell of a cell array. The syntax for indexing into field F of a structure array that resides in a cell of array C is

```
X = C{CellArrayIndex}(StructArrayIndex).F(FieldArrayIndex);
```

For example, row 1, column 3 of cell array A contains a structure array. Use A{1,3} to select this cell, and .name to display the field name for all elements of the structure array:

```
A{1,3}.name
ans =
    Ann Lane
ans =
    John Doe
```

To display all fields of a particular element of the structure array, type

```
A{1,3}(2)
ans =
       name: 'John Doe'
    billing: 139.7200
       test: [4.6228 17.8260 0.3701 12.3086 14.7641]
```

The test field of this structure array contains a 1-by-5 numeric array. Access the odd numbered elements of this field in the second element of the structure array:

```
A{1,3}(2).test(1:2:end)
ans =
    4.6228    0.3701   14.7641
```

Working With Cell Arrays Within Cells

The syntax for indexing into a cell array that resides in a cell of array C using content indexing is shown below. To use cell indexing on the inner cell array, replace the curly brace operator enclosing the InnerCellArrayIndes with parentheses.

The syntax for content indexing is

```
X = C{OuterCellArrayIndex}{InnerCellArrayIndex}
```

In the example cell array created at the start of this section, A{2,2} is a cell array that resides in a cell of the outer array A. To get the third row of the inner cell array, type

```
A{2,2}{3,:}
```

```
ans =
     7
ans =
     2
ans =
     9
ans =
     2
```

Note that MATLAB returns a comma-separated list. To have MATLAB return the list of elements as a vector instead, surround the previous expression with square brackets:

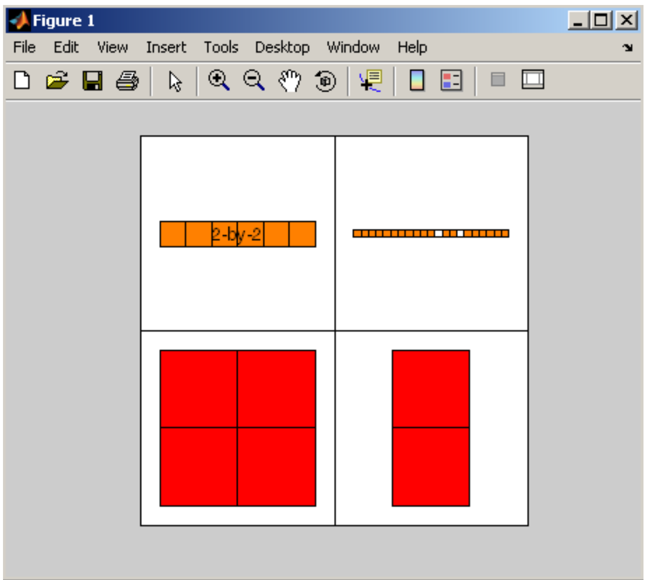
```
[A{2,2}{3,:}]
ans =
     7     2     9     2
```

Plotting the Cell Array

For a high-level graphical display of cell architecture, use the `cellplot` function. Consider a 2-by-2 cell array containing two text strings, a matrix, and a vector:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces this figure:



Deleting Cells

You can delete an entire dimension of cells using a single statement. Like standard array deletion, use vector subscripting when deleting a row or column of cells and assign the empty matrix to the dimension:

```
A(cell_subscripts) = []
```

When deleting cells, curly braces do not appear in the assignment statement at all.

Reshaping Cell Arrays

Like other arrays, you can reshape cell arrays using the `reshape` function. The number of cells must remain the same after reshaping; you cannot use `reshape` to add or remove cells:

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

size(A)
ans =
```

```
    3     4

B = reshape(A, 6, 2);

size(B)
ans =
    6     2
```

Replacing Lists of Variables with Cell Arrays

Cell arrays can replace comma-separated lists of MATLAB variables in

- Function input lists
- Function output lists
- Display operations
- Array constructions (square brackets and curly braces)

If you use the colon to index multiple cells in conjunction with the curly brace notation, MATLAB treats the contents of each cell as a separate variable. For example, assume you have a cell array `T` where each cell contains a separate vector. The expression `T{1:5}` is equivalent to a comma-separated list of the vectors in the first five cells of `T`.

Consider the cell array `C`:

```
C(1) = {[1 2 3]};
C(2) = {[1 0 1]};
C(3) = {1:10};
C(4) = {[9 8 7]};
C(5) = {3};
```

To convolve the vectors in `C(1)` and `C(2)` using `conv`,

```
d = conv(C{1:2})
d =
     1     2     4     2     3
```

Display vectors two, three, and four with

```
C{2:4}
ans =
     1     0     1

ans =
     1     2     3     4     5     6     7     8     9    10

ans =
     9     8     7
```

Similarly, you can create a new numeric array using the statement

```
B = [C{1}; C{2}; C{4}]
B =
     1     2     3
     1     0     1
     9     8     7
```

You can also use content indexing on the left side of an assignment to create a new cell array where each cell represents a separate output argument:

```
[D{1:2}] = eig(B)
D =
    [3x3 double]    [3x3 double]
```

You can display the actual eigenvectors and eigenvalues using `D{1}` and `D{2}`.

Note The `varargin` and `varargout` arguments allow you to specify variable numbers of input and output arguments for MATLAB functions that you create. Both `varargin` and `varargout` are cell arrays, allowing them to hold various sizes and kinds of MATLAB data. See "Passing Variable Numbers of Arguments" on page 4-34 in the MATLAB Programming documentation for details.

Applying Functions and Operators

Use indexing to apply functions and operators to the contents of cells. For example, use content indexing to call a function with the contents of a single cell as an argument:

```
A{1,1} = [1 2; 3 4];
A{1,2} = randn(3, 3);
A{1,3} = 1:5;

B = sum(A{1,1})
B =
     4     6
```

To apply a function to several cells of an unnested cell array, use a loop:

```
for k = 1:length(A)
    M{k} = sum(A{1,k});
end
```

Organizing Data in Cell Arrays

Cell arrays are useful for organizing data that consists of different sizes or kinds of data. Cell arrays are better than structures for applications where

- You need to access multiple fields of data with one statement.
- You want to access subsets of the data as comma-separated variable lists.
- You don't have a fixed set of field names.
- You routinely remove fields from the structure.

As an example of accessing multiple fields with one statement, assume that your data consists of

- A 3-by-4 array consisting of measurements taken for an experiment.
- A 15-character string containing a technician's name.
- A 3-by-4-by-5 array containing a record of measurements taken for the past five experiments.

For many applications, the best data construct for this data is a structure. However, if you routinely access only the first two fields of information, then a cell array might be more convenient for indexing purposes.

This example shows how to access the first and second elements of the cell array `TEST`:

```
[newdata,name] = deal(TEST{1:2})
```

This example shows how to access the first and second elements of the structure TEST:

```
newdata = TEST.measure
name = TEST.name
```

The varargin and varargout arguments are examples of the utility of cell arrays as substitutes for comma-separated lists. Create a 3-by-3 numeric array A:

```
A = [0 1 2; 4 0 7; 3 1 2];
```

Now apply the normest (2-norm estimate) function to A, and assign the function output to individual cells of B:

```
[B{1:2}] = normest(A)
B =
    [8.8826]    [4]
```

All of the output values from the function are stored in separate cells of B. B(1) contains the norm estimate; B(2) contains the iteration count.

Nesting Cell Arrays

A cell can contain another cell array, or even an array of cell arrays. (Cells that contain noncell data are called *leaf cells*.) You can use nested curly braces, the cell function, or direct assignment statements to create nested cell arrays. You can then access and manipulate individual cells, subarrays of cells, or cell elements.

Building Nested Arrays with Nested Curly Braces

You can nest pairs of curly braces to create a nested cell array. For example,

```
clear A
A(1,1) = {magic(5)};

A(1,2) = {{[5 2 8; 7 3 0; 6 7 3] 'Test 1'; [2-4i 5+7i] {17 []}}}
A =
```

```
    [5x5 double]    {2x2 cell}
```

Note that the right side of the assignment is enclosed in two sets of curly braces. The first set represents cell (1,2) of cell array A. The second "packages" the 2-by-2 cell array inside the outer cell.

Building Nested Arrays with the cell Function

To nest cell arrays with the cell function, assign the output of cell to an existing cell:

1 Create an empty 1-by-2 cell array.

```
A = cell(1,2);
```

2 Create a 2-by-2 cell array inside A(1,2).

```
A(1,2) = {cell(2,2)};
```

3 Fill A, including the nested array, using assignments.

```
A(1,1) = {magic(5)};
A{1,2}(1,1) = {[5 2 8; 7 3 0; 6 7 3]};
A{1,2}(1,2) = {'Test 1'};
A{1,2}(2,1) = {[2-4i 5+7i]};
A{1,2}(2,2) = {cell(1, 2)}
A{1,2}{2,2}(1) = {17};
```

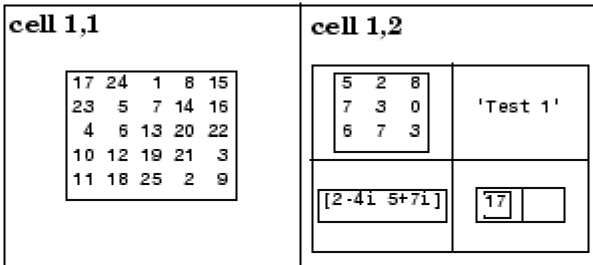
Note the use of curly braces until the final level of nested subscripts. This is required because you need to access cell contents to access cells within cells.

You can also build nested cell arrays with direct assignments using the statements shown in step 3 above.

Indexing Nested Cell Arrays

To index nested cells, concatenate indexing expressions. The first set of subscripts accesses the top layer of cells, and subsequent sets of parentheses access successively deeper layers.

For example, array A has three levels of nesting:



- To access the 5-by-5 array in cell (1,1), use A{1,1}.
- To access the 3-by-3 array in position (1,1) of cell (1,2), use A{1,2}{1,1}.
- To access the 2-by-2 cell array in cell (1,2), use A{1,2}.
- To access the empty cell in position (2,2) of cell (1,2), use A{1,2}{2,2}{1,2}.

Converting Between Cell and Numeric Arrays

Use for loops to convert between cell and numeric formats. For example, create a cell array F:

```
F{1,1} = [1 2; 3 4];
F{1,2} = [-1 0; 0 1];
F{2,1} = [7 8; 4 1];
F{2,2} = [4i 3+2i; 1-8i 5];
```

Now use three for loops to copy the contents of F into a numeric array NUM:

```
for k = 1:4
   for m = 1:2
      for n = 1:2
         NUM(m,n,k) = F{k}(m,n);
      end
   end
end
```

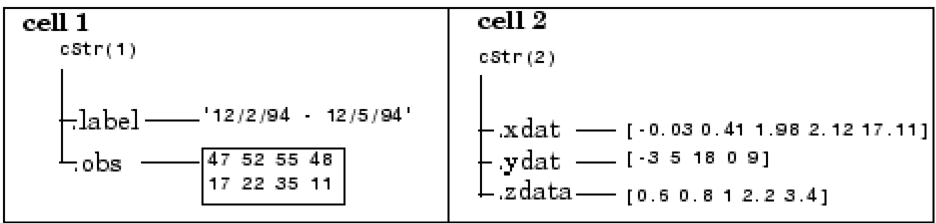
Similarly, you must use for loops to assign each value of a numeric array to a single cell of a cell array:

```
G = cell(1,16);
for m = 1:16
   G{m} = NUM(m);
end
```

Cell Arrays of Structures

Use cell arrays to store groups of structures with different field architectures:

```
cStr = cell(1,2);
cStr{1}.label = '12/2/94 - 12/5/94';
cStr{1}.obs = [47 52 55 48; 17 22 35 11];
cStr{2}.xdata = [-0.03 0.41 1.98 2.12 17.11];
cStr{2}.ydata = [-3 5 18 0 9];
cStr{2}.zdata = [0.6 0.8 1 2.2 3.4];
```



Cell 1 of the `cStr` array contains a structure with two fields, one a string and the other a vector. Cell 2 contains a structure with three vector fields.

When building cell arrays of structures, you must use content indexing. Similarly, you must use content indexing to obtain the contents of structures within cells. The syntax for content indexing is

```
cellArray{index}.field
```

For example, to access the `label` field of the structure in cell 1, use `cStr{1}.label`.

Function Summary

This table describes the MATLAB functions for working with cell arrays.

Function	Description
`cell`	Create a cell array.
`cell2struct`	Convert a cell array into a structure array.
`celldisp`	Display cell array contents.
`cellfun`	Apply a cell function to a cell array.
`cellplot`	Display a graphical depiction of a cell array.
`deal`	Copy input to separate outputs.
`iscell`	Return `true` for a cell array.
`num2cell`	Convert a numeric array into a cell array.

Function Handles

In this section...

"Overview" on page 2-115

"Constructing and Invoking a Function Handle" on page 2-115

"Calling a Function Using Its Handle" on page 2-116

"Simple Function Handle Example" on page 2-116

Overview

A *function handle* is a MATLAB value and data type that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics® callbacks).

Read more about function handles in the section, "Function Handles" on page 4-22.

Constructing and Invoking a Function Handle

You construct a handle for a specific function by preceding the function name with an @ sign. Use only the function *name* (with no path information) after the @ sign:

```
fhandle = @functionname
```

Handles to Anonymous Functions

Another way to construct a function handle is to create an anonymous function. For example,

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument x. The variable sqr contains a handle to the anonymous function. See "Anonymous Functions" on page 5-3 for more information.

Calling a Function Using Its Handle

To execute a function associated with a function handle, use the syntax shown here, treating the function handle `fhandle` as if it were a function name:

```
fhandle(arg1, arg2, ..., argn)
```

If the function being called takes no input arguments, then use empty parentheses after the function handle name:

```
fhandle()
```

Simple Function Handle Example

The following example calls a function `plotFHandle`, passing it a handle for the MATLAB `sin` function. `plotFHandle` then calls the `plot` function, passing it some data and the function handle to `sin`. The plot function calls the function associated with the handle to compute its y-axis values:

```
function x = plotFHandle(fhandle, data)
plot(data, fhandle(data))
```

Call `plotFhandle` with a handle to the `sin` function and the value shown below:

```
plotFHandle(@sin, -pi:0.01:pi)
```

MATLAB Classes

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the M-file functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that override existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

Read more about MATLAB classes in Chapter 9, "Classes and Objects".

Java Classes

MATLAB provides an interface to the Java programming language that enables you to create objects from Java classes and call Java methods on these objects. A Java class is a MATLAB data type. Native and third-party classes are already available through the MATLAB interface. You can also create your own Java class definitions and bring them into MATLAB.

The MATLAB Java interface enables you to

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking
- Access third-party Java classes
- Easily construct Java objects in MATLAB
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

Read more about Java classes in MATLAB in “Calling Java from MATLAB” in the MATLAB External Interfaces documentation.

Basic Program Components

Variables (p. 3-2)	Guidelines for creating variables; global and persistent variables; variable scope and lifetime
Keywords (p. 3-13)	Reserved words that you should avoid using
Special Values (p. 3-14)	Functions that return constant values, like `pi` or `inf`
Operators (p. 3-16)	Arithmetic, relational, and logical operators
MATLAB Expressions (p. 3-27)	Executing user-supplied strings; constructing executable strings, shell escape functions
Regular Expressions (p. 3-30)	A versatile way to search and replace character strings
Comma-Separated Lists (p. 3-79)	Using lists with structures and cell arrays to simplify your code
Program Control Statements (p. 3-87)	Using statements such as `if`, `for`, and `try-catch` to control the code path your program follows
Symbol Reference (p. 3-96)	Using statements such as `if`, `for`, and `try-catch` to control the code path your program follows
Internal MATLAB Functions (p. 3-108)	Description of the M-file, built-in, and overloaded function types supplied with MATLAB

Variables

In this section…

"Types of Variables" on page 3-2

"Naming Variables" on page 3-6

"Guidelines to Using Variables" on page 3-10

"Scope of a Variable" on page 3-10

"Lifetime of a Variable" on page 3-12

Types of Variables

A MATLAB variable is essentially a tag that you assign to a value while that value remains in memory. The tag gives you a way to reference the value in memory so that your programs can read it, operate on it with other data, and save it back to memory.

MATLAB provides three basic types of variables:

- "Local Variables" on page 3-2
- "Global Variables" on page 3-3
- "Persistent Variables" on page 3-5

Local Variables

Each MATLAB function has its own local variables. These are separate from those of other functions (except for nested functions), and from those of the base workspace. Variables defined in a function do not remain in memory from one function call to the next, unless they are defined as `global` or `persistent`.

Scripts, on the other hand, do not have a separate workspace. They store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace.

Note If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

Global Variables

If several functions, and possibly the base workspace, all declare a particular name as `global`, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose, for example, you want to study the effect of the interaction coefficients, α and β, in the Lotka-Volterra predator-prey model.

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

Create an M-file, `lotka.m`.

```
function yp = lotka(t,y)
%LOTKA   Lotka-Volterra predator-prey model.
global ALPHA BETA
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA
ALPHA = 0.01
BETA = 0.02
[t,y] = ode23(@lotka,[0,10],[1; 1]);
plot(t,y)
```

The two `global` statements make the values assigned to `ALPHA` and `BETA` at the command prompt available inside the function defined by `lotka.m`. They can be modified interactively and new solutions obtained without editing any files.

Creating Global Variables. Each function that uses a global variable must first declare the variable as `global`. It is usually best to put global declarations toward the beginning of the function. You would declare global variable `MAXLEN` as follows:

```
global MAXLEN
```

If the M-file contains subfunctions as well, then each subfunction requiring access to the global variable must declare it as `global`. To access the variable from the MATLAB command line, you must declare it as `global` at the command line.

MATLAB global variable names are typically longer and more descriptive than local variable names, and often consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code, and to reduce the chance of accidentally redefining a global variable.

Displaying Global Variables. To see only those variables you have declared as global, use the `who` or `whos` functions with the literal, `global`.

```
global MAXLEN MAXWID
MAXLEN = 36;  MAXWID = 78;
len = 5;  wid = 21;

whos global
  Name         Size                    Bytes  Class

  MAXLEN       1x1                         8  double array (global)
  MAXWID       1x1                         8  double array (global)

Grand total is 2 elements using 16 bytes
```

Suggestions for Using Global Variables. A certain amount of risk is associated with using global variables and, because of this, it is recommended that you use them sparingly. You might, for example, unintentionally give a global variable in one function a name that is already used for a global variable in another function. When you run your application, one function may overwrite the variable used by the other. This error can be difficult to track down.

Another problem comes when you want to change the variable name. To make a change without introducing an error into the application, you must find every occurrence of that name in your code (and other people's code, if you share functions).

Alternatives to Using Global Variables. Instead of using a global variable, you may be able to

- Pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

  If this means that you have to pass a number of additional variables, you can put them into a structure or cell array and just pass it as one additional argument.

- Use a persistent variable (described in the next section), if you only need to make the variable persist in memory from one function call to the next.

Persistent Variables

Characteristics of persistent variables are

- You can declare and use them within M-file functions only.
- Only the function in which the variables are declared is allowed access to it.
- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.

You must declare persistent variables before you can use them in a function. It is usually best to put your persistent declarations toward the beginning of the function. You would declare persistent variable `SUM_X` as follows:

```
persistent SUM_X
```

If you clear a function that defines a persistent variable (i.e., using `clear` *`functionname`* or `clear all`), or if you edit the M-file for that function, MATLAB clears all persistent variables used in that function.

You can use the `mlock` function to keep an M-file from being cleared from memory, thus keeping persistent variables in the M-file from being cleared as well.

Initializing Persistent Variables. When you declare a persistent variable, MATLAB initializes its value to an empty matrix, []. After the declaration statement, you can assign your own value to it. This is often done using an isempty statement, as shown here:

```
function findSum(inputvalue)
persistent SUM_X

if isempty(SUM_X)
   SUM_X = 0;
end
SUM_X = SUM_X + inputvalue
```

This initializes the variable to 0 the first time you execute the function, and then it accumulates the value on each iteration.

Naming Variables

MATLAB variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so A and a are not the same variable.

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function namelengthmax), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
N =
    63
```

The genvarname function can be useful in creating variable names that are both valid and unique. See the genvarname reference page to find out how to do this.

Verifying a Variable Name

You can use the `isvarname` function to make sure a name is valid before you use it. `isvarname` returns 1 if the name is valid, and 0 otherwise.

```
isvarname 8th_column
ans =
     0                    % Not valid - begins with a number
```

Avoid Using Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name, either one of your own M-file functions or one of the functions in the MATLAB language. If you define a variable with a function name, you will not be able to call that function until you either remove the variable from memory with the `clear` function, or invoke the function using `builtin`.

For example, if you enter the following command, you will not be able to use the MATLAB `disp` function until you clear the variable with `clear disp`.

```
disp = 50;
```

To test whether a proposed variable name is already used as a function name, use

```
which -all variable_name
```

Potential Conflict with Function Names

There are some MATLAB functions that have names that are commonly used as variable names in programming code. A few examples of such functions are `i`, `j`, `mode`, `char`, `size`, and `path`.

If you need to use a variable that is also the name of a MATLAB function, and have determined that you have no need to call the function, you should be aware that there is still a possibility for conflict. See the following two examples:

- "Variables Loaded From a MAT-File" on page 3-8
- "Variables In Evaluation Statements" on page 3-9

Variables Loaded From a MAT-File. The function shown below loads previously saved data from MAT-file settings.mat. It is supposed to display the value of one of the loaded variables, mode. However, mode is also the name of a MATLAB function and, in this case, MATLAB interprets it as the function and not the variable loaded from the MAT-file:

```
function show_mode
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Assume that mode already exists in the MAT-file. Execution of the function shows that, even though mode is successfully loaded into the function workspace as a variable, when MATLAB attempts to operate on it in the last line, it interprets mode as a function. This results in an error:

```
show_mode
  Name       Size                     Bytes  Class

  mode       1x6                         12  char array

Grand total is 6 elements using 12 bytes

??? Error using ==> mode
Not enough input arguments.

Error in ==> show_mode at 4
fprintf('Mode is set to %s\n', mode)
```

Because MATLAB parses function M-files before they are run, it needs to determine before runtime which identifiers in the code are variables and which are functions. The function in this example does not establish mode as a variable name and, as a result, MATLAB interprets it as a function name instead.

There are several ways to make this function work as intended without having to change the variable name. Both indicate to MATLAB that the name represents a variable, and not a function:

- Name the variable explicitly in the load statement:

```
function show_mode
load settings mode;
whos mode
fprintf('Mode is set to %s\n', mode)
```

- Initialize the variable (e.g., set it to an empty matrix or empty string) at the start of the function:

```
function show_mode
mode = '';
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Variables In Evaluation Statements. Variables used in evaluation statements such as `eval`, `evalc`, and `evalin` can also be mistaken for function names. The following M-file defines a variable named `length` that conflicts with MATLAB `length` function:

```
function find_area
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

The second line of this code would seem to establish `length` as a variable name that would be valid when used in the statement on the third line. However, when MATLAB parses this line, it does not consider the *contents* of the string that is to be evaluated. As a result, MATLAB has no way of knowing that `length` was meant to be used as a variable name in this program, and the name defaults to a function name instead, yielding the following error:

```
find_area
??? Error using ==> length
Not enough input arguments.
```

To force MATLAB to interpret `length` as a variable name, use it in an explicit assignment statement first:

```
function find_area
length = [];
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

Guidelines to Using Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in M-files:

- You do not need to type or declare variables used in M-files (with the possible exception of designating them as `global` or `persistent`).
- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

Scope of a Variable

MATLAB stores variables in a part of memory called a workspace. The *base workspace* holds variables created during your interactive MATLAB session and also any variables created by running M-file scripts. Variables created at the MATLAB command prompt can also be used by scripts without having to declare them as global.

Functions do not use the base workspace. Every function has its own *function workspace*. Each function workspace is kept separate from the base workspace and all other workspaces to protect the integrity of the data used by that function. Even subfunctions that are defined in the same M-file have a separate function workspace.

Extending Variable Scope

In most cases, variables created within a function are known only within that function. These variables are not available at the MATLAB command prompt or to any other function or subfunction.

Passing Variables from Another Workspace. The most secure way to extend the scope of a function variable is to pass it to other functions as an argument in the function call. Since MATLAB passes data only by value, you also need to add the variable to the return values of any function that modifies its value.

Evaluating in Another Workspace Using evalin. Functions can also obtain variables from either the base or the caller's workspace using the `evalin` function. The example below, `compareAB_1`, evaluates a command in the context of the MATLAB command line, taking the values of variables `A` and `B` from the base workspace.

Define `A` and `B` in the base workspace:

```
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Use `evalin` to evaluate the command `A(find(A<=B))` in the context of the MATLAB base workspace:

```
function C = compareAB_1
C = evalin('base', 'A(find(A<=B))');
```

Call the function. You do not have to pass the variables because they are made available to the function via the `evalin` function:

```
C = compareAB_1
C =
    13     9    15
```

You can also evaluate in the context of the caller's workspace by specifying `'caller'` (instead of `'base'`) as the first input argument to `evalin`.

Using Global Variables. A third way to extend variable scope is to declare the variable as `global` within every function that needs access to it. If you do this, you need make sure that no functions with access to the variable overwrite its value unintentionally. For this reason, it is recommended that you limit the use of global variables.

Create global vectors `A` and `B` in the base workspace:

```
global A
global B
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Also declare them in the function to be called:

```
function C = compareAB_2
global A
```

```
global B

C = A(find(A<=B));
```

Call the function. Again, you do not have to pass A and B as arguments to the called function:

```
C = compareAB_2
C =
    13     9    15
```

Scope in Nested Functions

Variables within nested functions are accessible to more than just their immediate function. As a general rule, the scope of a local variable is the largest containing function body in which the variable appears, and all functions nested within that function. For more information on nested functions, see “Variable Scope in Nested Functions” on page 5-19.

Lifetime of a Variable

Variables created at the MATLAB command prompt or in an M-file script exist until you clear them or end your MATLAB session. Variables in functions exist only until the function completes unless they have been declared as `global` or `persistent`.

Keywords

MATLAB reserves certain words for its own use as keywords of the language. To list the keywords, type

```
iskeyword
ans =
    'break'
    'case'
    'catch'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```

See the online function reference pages to learn how to use these keywords.

You should not use MATLAB keywords other than for their intended purpose. For example, a keyword should not be used as follows:

```
while = 5;
??? while = 5;
          |
Error: Expected a variable, function, or constant, found "=".
```

Special Values

Several functions return important special values that you can use in your M-files.

Function	Return Value
`ans`	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in `ans`.
`eps`	Floating-point relative accuracy. This is the tolerance MATLAB uses in its calculations.
`intmax`	Largest 8-, 16-, 32-, or 64-bit integer your computer can represent.
`intmin`	Smallest 8-, 16-, 32-, or 64-bit integer your computer can represent.
`realmax`	Largest floating-point number your computer can represent.
`realmin`	Smallest positive floating-point number your computer can represent.
`pi`	`3.1415926535897...`
`i, j`	Imaginary unit.
`inf`	Infinity. Calculations like `n/0`, where `n` is any nonzero real value, result in `inf`.
`NaN`	Not a Number, an invalid numeric value. Expressions like `0/0` and `inf/inf` result in a `NaN`, as do arithmetic operations involving a `NaN`. Also, if `n` is complex with a zero real part, then `n/0` returns a value with a `NaN` real part.
`computer`	Computer type.
`version`	MATLAB version string.

Here are some examples that use these values in MATLAB expressions.

```
x = 2 * pi
x =
   6.2832

A = [3+2i 7-8i]
A =
   3.0000 + 2.0000i   7.0000 - 8.0000i

tol = 3 * eps
tol =
   6.6613e-016

intmax('uint64')
ans =
   18446744073709551615
```

Operators

In this section...

"Arithmetic Operators" on page 3-16

"Relational Operators" on page 3-17

"Logical Operators" on page 3-19

"Operator Precedence" on page 3-25

Arithmetic Operators

Arithmetic operators perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power. The following table provides a summary. For more information, see the arithmetic operators reference page.

Operator	Description
+	Addition
-	Subtraction
.*	Multiplication
./	Right division
.\	Left division
+	Unary plus
-	Unary minus
:	Colon operator
.^	Power
.'	Transpose
'	Complex conjugate transpose
*	Matrix multiplication
/	Matrix right division

Operator	Description
\	Matrix left division
^	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

3 * A
ans =
    24     3    18
     9    15    21
    12    27     6
```

Relational Operators

Relational operators compare operands quantitatively, using operators like "less than" and "not equal to." The following table provides a summary. For more information, see the relational operators reference page.

Operator	Description
<	Less than
<=	Less than or equal to

Operator	Description
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Relational Operators and Arrays

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of A is equal to the corresponding element of B.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4 -1 7];

A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive logical 1. Locations where the relation is false receive logical 0.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []
```

return an error if A is not 0-by-0 or 1-by-1. This behavior is consistent with that of all other binary operators, such as +, -, >, <, &, |, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operators and functions:

- Element-wise — operate on corresponding elements of logical arrays.
- Bit-wise — operate on corresponding bits of integer values or arrays.
- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform elementwise logical operations on their inputs to produce a like-sized output array.

The examples shown in the following table use vector inputs `A` and `B`, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
`&`	Returns `1` for every element location that is `true` (nonzero) in both arrays, and `0` for all other elements.	`A & B = 01001`
`|`	Returns `1` for every element location that is `true` (nonzero) in either one or the other, or both arrays, and `0` for all other elements.	`A | B = 11101`
`~`	Complements each element of the input array, `A`.	`~A = 10010`
`xor`	Returns `1` for every element location that is `true` (nonzero) in only one array, and `0` for all other elements.	`xor(A,B) = 10100`

For operators and functions that take two array operands, (&, |, and xor), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical 1, or true.

Operator Overloading. You can overload the &, |, and ~ operators to make their behavior dependent upon the data type on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
A & B	and(A, B)
A | B	or(A, B)
~A	not(A)

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an elementwise fashion, are any and all. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, any and all operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional dimension input to operate on a specific dimension of the array.

The examples shown in the following table use array input A, where

```
A = [0   1   2;
     0  -3   8;
     0   5   0];
```

Function	Description	Example
`any(A)`	Returns `1` for a vector where *any* element of the vector is `true` (nonzero), and `0` if no elements are `true`.	`any(A) ans = 0 1 1`
`all(A)`	Returns `1` for a vector where *all* elements of the vector are `true` (nonzero), and `0` if all elements are not `true`.	`all(A) ans = 0 1 0`

Note The `all` and `any` functions ignore any `NaN` values in the input arrays.

Short-Circuiting in Elementwise Operators. When used in the context of an `if` or `while` expression, and only in this context, the elementwise `|` and `&` operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

So, although the statement `1|[]` evaluates to `false`, the same statement evaluates to `true` when used in either an `if` or `while` expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
   The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to `false`

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between `A` and `B`,

```
A = [1 1];   B = [2 0 1];
A|B        % This generates an error.
```

works within the context of an `if` or `while` expression:

```
if (A|B) disp 'The statement is true',  end;
```

```
The statement is true
```

Logical Expressions Using the find Function. The `find` function determines the indices of array elements that meet a given logical condition. The function is useful for creating masks and index matrices. In its most general form, `find` returns a single vector of indices. This vector can be used to index into arrays of any size or shape.

For example,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

i = find(A > 8);
A(i) = 100
A =
   100     2     3   100
     5   100   100     8
   100     7     6   100
     4   100   100     1
```

Note An alternative to using `find` in this context is to index into the matrix using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16     2     3    13
```

```
     5    11    10     8
     9     7     6    12
     4    14    15     1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4
```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs A and B, where

```
A = 28;                 % binary 11100
B = 21;                 % binary 10101
```

Function	Description	Example
bitand	Returns the bit-wise AND of two nonnegative integer arguments.	bitand(A,B) = 20 (binary 10100)
bitor	Returns the bit-wise OR of two nonnegative integer arguments.	bitor(A,B) = 29 (binary 11101)

Function	Description	Example
bitcmp	Returns the bit-wise complement as an n-bit number, where n is the second input argument to bitcmp.	bitcmp(A,5) = 3 (binary 00011)
bitxor	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	bitxor(A,B) = 9 (binary 01001)

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
&&	Returns logical 1 (true) if both inputs evaluate to true, and logical 0 (false) if they do not.
||	Returns logical 1 (true) if either input, or both, evaluate to true, and logical 0 (false) if they do not.

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

If A equals zero, then the entire expression will evaluate to logical 0 (false), regardless of the value of B. Under these circumstances, there is no need to evaluate B because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is true. Again, regardless of the value of B, the statement will evaluate to true. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute an M-file function only if the M-file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, `myfun.m`, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids divide-by-zero errors when `b` equals zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the `&&` and `||` operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1. Parentheses ()
2. Transpose (`.'`), power (`.^`), complex conjugate transpose (’), matrix power (`^`)
3. Unary plus (+), unary minus (`-`), logical negation (~)
4. Multiplication (`.*`), right division (`./`), left division (`.\`), matrix multiplication (`*`), matrix right division (`/`), matrix left division (`\`)
5. Addition (+), subtraction (`-`)
6. Colon operator (`:`)

7. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)

8. Element-wise AND (`&`)

9. Element-wise OR (`|`)

10. Short-circuit AND (`&&`)

11. Short-circuit OR (`||`)

Precedence of AND and OR Operators

MATLAB always gives the `&` operator precedence over the `|` operator. Although MATLAB typically evaluates expressions from left to right, the expression `a|b&c` is evaluated as `a|(b&c)`. It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of `&` and `|`.

The same precedence rule holds true for the `&&` and `||` operators.

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this example:

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2
C =
    0.7500    9.0000    0.2000

C = (A./B).^2
C =
    2.2500   81.0000    1.0000
```

MATLAB Expressions

In this section...

"String Evaluation" on page 3-27

"Shell Escape Functions" on page 3-28

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The `eval` function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the `eval` syntax is

```
eval('string')
```

For example, this code uses `eval` on an expression to generate a Hilbert matrix of order `n`.

```
t = '1/(m + n - 1)';
for m = 1:k
    for n = 1:k
        a(m,n) = eval(t);
    end
end
```

Here is an example that uses `eval` on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation. You can concatenate strings to create a complete expression for input to `eval`. This code shows how `eval` can create 10 variables named `P1`, `P2`, `...`, `P10`, and set each of them to a different value.

```
for n = 1:10
    eval(['P', int2str(n), '= n .^ 2'])
```

```
end
```

feval

The `feval` function differs from `eval` in that it executes a function rather than a MATLAB expression. The function to be executed is specified in the first argument by either a function handle or a string containing the function name.

You can use `feval` and the `input` function to choose one of several tasks defined by M-files. This example uses function handles for the `sin`, `cos`, and `log` functions.

```
fun = {@sin; @cos; @log};
k = input('Choose function number: ');
x = input('Enter value: ');
feval(fun{k}, x)
```

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command `!` to make external stand-alone programs act like new MATLAB functions. A shell escape M-function is an M-file that

1. Saves the appropriate variables on disk.
2. Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).
3. Loads the processed file back into the workspace.

For example, look at the code for `garfield.m`, below. This function uses an external function, `gareqn`, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This M-file

1. Saves the input arguments `a`, `b`, `q`, and `r` to a MAT-file in the workspace using the `save` command.
2. Uses the shell escape operator to access a C or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.
3. Loads the `gardata` MAT-file described in “Using MAT-Files” to obtain the results.

Regular Expressions

In this section...

"Overview" on page 3-30
"MATLAB Regular Expression Functions" on page 3-31
"Elements of an Expression" on page 3-32
"Character Classes" on page 3-33
"Character Representation" on page 3-36
"Grouping Operators" on page 3-37
"Nonmatching Operators" on page 3-39
"Positional Operators" on page 3-39
"Lookaround Operators" on page 3-40
"Quantifiers" on page 3-45
"Tokens" on page 3-48
"Named Capture" on page 3-53
"Conditional Expressions" on page 3-55
"Dynamic Regular Expressions" on page 3-57
"String Replacement" on page 3-66
"Handling Multiple Strings" on page 3-68
"Operator Summary" on page 3-71

Overview

A regular expression is a string of characters that defines a certain pattern. You would normally use a regular expression in searching through text for a group of words that matches this pattern, perhaps while parsing program input, or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any

number of non-whitespace characters (indicated by `'\w*'`). This pattern matches any of the following:

```
Jon, John, Jonathan, Johnny
```

MATLAB supports most of the special characters, or *metacharacters*, commonly used with regular expressions and provides several functions to use in searching and replacing text with these expressions.

MATLAB Regular Expression Functions

Several MATLAB functions support searching and replacing characters using regular expressions:

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.
`regexptranslate`	Translate string into regular expression.

See the function reference pages to obtain more information on these functions. For more information on how to use regular expressions in general, consult a reference on that subject.

The `regexp` and `regexpi` functions return up to six outputs in the order shown in the reference page for `regexp`. You can select specific outputs to be returned by using one or more of the following qualifiers with these commands:

Qualifier	Value Returned
`'start'`	Starting index of each substring matching the expression
`'end'`	Ending index of each substring matching the expression
`'tokenExtents'`	Starting and ending indices of each substring matching a token in the expression
`'match'`	Text of each substring matching the expression

Qualifier	Value Returned
'tokens'	Text of each token captured
'names'	Name and text of each named token captured
'split'	Treating each match as a delimiter, the text of each substring between such delimiters.

There is an additional qualifier named `'once'` that you can use to return only the first match found.

Elements of an Expression

Tables and examples in the sections that follow show the metacharacters and syntax supported by the `regexp`, `regexpi`, and `regexprep` functions in MATLAB. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regexptranslate` function.

These elements are presented under these categories:

- "Character Classes" on page 3-33
- "Character Representation" on page 3-36
- "Grouping Operators" on page 3-37
- "Nonmatching Operators" on page 3-39
- "Positional Operators" on page 3-39
- MATLAB Programming on page 1
- "Quantifiers" on page 3-45
- "Tokens" on page 3-48
- "Named Capture" on page 3-53
- "Conditional Expressions" on page 3-55
- "Dynamic Regular Expressions" on page 3-57

Each table is followed by a set of examples that show how to use the syntax presented in that table.

Character Classes

Character classes represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., non-whitespace).

Operator	Usage
`.`	Any single character, including white space
[$c_1c_2c_3$]	Any character contained within the brackets: c_1 or c_2 or c_3
[^$c_1c_2c_3$]	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
[c_1-c_2]	Any character in the range of c_1 through c_2
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`
`\S`	Any non-whitespace character; equivalent to `[^ \f\n\r\t\v]`
`\w`	Any alphabetic, numeric, or underscore character; equivalent to `[a-zA-Z_0-9]`. (This does not apply to non-English character sets).
`\W`	Any character that is not alphabetic, numeric, or underscore; equivalent to `[^a-zA-Z_0-9]`. (True only for English character sets).
`\d`	Any numeric digit; equivalent to `[0-9]`
`\D`	Any nondigit character; equivalent to `[^0-9]`

The following examples demonstrate how to use the character classes listed above. See the `regexp` reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

Use `'..ain'` in an expression to match a sequence of five characters ending in `'ain'`. Note that `.` matches white-space characters as well:

```
regexp(str, '..ain')
ans =
     4    13    24    39
```

Matches `' rain'`, `'Spain'`, `' main'`, and `'plain'`.

Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier `'match'`. In this case, `regexp` returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    ' rain'    'Spain'    ' main'    'plain'
```

Selected Characters — [c1c2c3]

Use [$c_1c_2c_3$] in an expression to match selected characters `r`, `p`, or `m` followed by `'ain'`. Specify two qualifiers this time, `'match'` and `'start'`, along with an output argument for each, `mat` and `idx`. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
     5    14    25
```

Range of Characters — [c1 - c2]

Use [c_1-c_2] in an expression to find words that begin with a letter in the range of A through Z:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'    'Spain'
idx =
     1    13
```

Word and White-Space Characters — \w, \s

Use \w and \s in an expression to find words that end with the letter n followed by a white-space character. Add a new qualifier, 'end', to return the str index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5    10    13    32
ix2 =
     9    12    18    34
```

Numeric Digits — \d

Use \d to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9    12    15
```

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
`\a`	Alarm (beep)
`\\`	Backslash
`\$`	Dollar sign
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Octal and Hexadecimal — \o, \x

Use `\x` and `\o` in an expression to find a comma (hex `2C`) followed by a space (octal `40`) followed by the character `2`:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\x2C\o{40}2', 'match', 'start')
mat =
    ', 2'
idx =
    10
```

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the `{2,}` multiplier to all three consecutive characters, you can first make a group of the characters and then apply the `(?:)` quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s?){2,}', 'match')
ans =
    'B5 A2 '    'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, `|`. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$expr_1$|$expr_2$	Match expression $expr_1$ or expression $expr_2$.

Grouping and Capture — (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see “Using Tokens — Example 1” on page 3-50.

Grouping Only — (?:expr)

Use `(?:expr)` to group a nonvowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome `pstr`. Specify at least

two consecutive occurrences ({2,}) of this group. Return the starting and ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[^aeiou][aeiou]){2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12    23    31
ix2 =
    15    26    34
```

Remove the grouping, and the {2,} now applies only to [aeiou]. The command is entirely different now as it looks for a nonvowel followed by at least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match — expr1|expr2

Use $p_1|p_2$ to pick out words in the string that start with let or tel:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

Nonmatching Operators

The `comment` operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'    'Norah'    'Sharon'
```

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Start and End of String Match — ^expr, expr$

Use `^expr` to match words starting with the letter `m` or `M` only when it begins the string, and `expr$` to match words ending with `m` or `M` only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
ans =
    'Marge'    'telegram'
```

Start and End of Word Match — \<expr, expr\>

Use `\<expr` to match any words starting with `n` or `N`, or ending with `e` or `E`:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'    'Norah'    'see'
```

Exact Word Match — \<expr\>

Use `\<expr\>` to match a word starting with an n or N and ending with an `h` or `H`:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
`(?=expr)`	Look ahead from current position and test if `expr` is found.
`(?!expr)`	Look ahead from current position and test if `expr` is not found

Operator	Usage
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (`\w*`) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters `vision`. The resulting match includes only that part of the string that matches the `\w*` operator; it does not include those characters that match the lookahead expression `(?=vision)`:

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
    14
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter `v`, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
       '\w*(?=vision).', 'match')
ans =
    'telev'
```

Note You can also use lookaround operators to perform a logical AND of two elements. See “Using Lookaround as a Logical Operator” on page 3-44.

Lookahead — expr(?=test)

Look ahead to the location of each of these national parks to identify those situated in Tanzania:

```
AfricanParks = {'Arusha, Tanzania', 'Limpopo, Mozambique', ...
   'Chobe, Botswana', 'Amboseli, Kenya', 'Mikumi, Tanzania', ...
   'Kabelaga, Uganda', 'Gonarezhou, Zimbabwe', ...
   'Uangudi, Ethiopia', 'Akagera, Rwanda', ...
   'Etosha, Namibia', 'Kilimanjaro, Tanzania', ...
   'Kasanga, Zambia', 'Udzungwa, Tanzania', 'Omo, Ethiopia'};

T = regexp(AfricanParks, '.*(?=, Tanzania)', 'match');
```

The result `T` is a cell array of empty and full character strings. Display the results:

```
for k=1:numel(AfricanParks)
   if k==1, disp 'Parks in Tanzania:', end
   if ~isempty(T{k})
      fprintf('   %s\n', char(T{k}))
   end
end

Parks in Tanzania:
   Arusha
   Mikumi
   Kilimanjaro
   Udzungwa
```

Negative Lookahead — expr(?!test)

Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
   5    6    7    8    9   10   11   12   13   14   15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```
regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
```

```
ans =
   05   06   07   08   09  10  11  12  13  14  15
```

Lookbehind — (?<=test)expr

This example uses the lookbehind operator to extract different types of currency from a string. Start by identifying the euro, British pound, Japanese yen, and American dollar symbols using their hexadecimal Unicode values:

```
euro = char(hex2dec('20AC'))
euro =
   €

pound = char(hex2dec('00A3'))
pound =
   £

yen = char(hex2dec('00A5'))
yen =
   ¥

dollar = char(hex2dec('0024'))
dollar =
   $
```

Compose a string of monetary values:

```
str = [euro '10.50 ' pound '6.94 ' yen '1649.40 ' dollar ...
      '13.67']
str =
   €10.50 £6.94 ¥1649.40 $13.67
```

Using `regexp`, match numeric and decimal point characters, but only if you can look behind and find the desired currency symbol immediately preceding those characters:

```
regexp(str, '(?<=\x{20AC})[\d\.]+', 'match')
ans =
    '10.50'
```

```
regexp(str, '(?<=\x{00A3})[\d\.]+', 'match')
ans =
    '6.94'

regexp(str, '(?<=\x{00A5})[\d\.]+', 'match')
ans =
    '1649.40'

regexp(str, '(?<=\x{0024})[\d\.]+', 'match')
ans =
    '13.67'
```

Negative Lookbehind — (?<!test)expr

Use `(?<!test)expr` to find all words that do not follow a comma and zero or more spaces:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat idx] = regexp(poestr, '(?<!,\s*\w*)\w*', 'match', 'start')
mat =
   'While' 'I' 'nodded' 'napping' 'there' 'came' 'a' 'tapping'
idx =
     1     7     9    24    42    48    53    55
```

Using Lookaround as a Logical Operator

You can use lookaround operators to perform a logical AND, as shown in this example. The expression used here finds all words that contain a sequence of two letters under the condition that the two letters are identical *and* are in the range a through m. (The expression `'(?=[a-m])'` is a lookahead test for the range a through m, and the expression `'(.)\1'` tests for identical characters using a token):

```
[mat idx] = regexp(poestr, '\<\w*(?=[a-m])(.)\1\w*\>', ...
   'match', 'start')
mat =
    'nodded'    'suddenly'
idx =
```

```
    9      33
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression `expr` *after* the test expression `test`:

```
(?=test)expr or (?!test)expr
```

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a `+` or `?` immediately following one of the six basic quantifying operators.

Operator	Usage
`expr{m,n}`	Must occur at least `m` times but no more than `n` times.
`expr{m,}`	Must occur at least `m` times.
`expr{n}`	Must match exactly `n` times. Equivalent to `{n,n}`.
`expr?`	Match the preceding element 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match the preceding element 0 or more times. Equivalent to `{0,}`.
`expr+`	Match the preceding element 1 or more times. Equivalent to `{1,}`.
`q_expr+`	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term `q_expr` represents any of the expressions shown in the top six rows of this table.
`q_expr?`	Match only as much of the quantified expression as necessary. The term `q_expr` represents any of the expressions shown in the top six rows of this table. For an example, see "Lazy Quantifiers — expr*?" on page 3-47, below.

Zero or One — expr?

Use `?` to make the HTML `<code>` and `</code>` tags optional in the string. The first string, `hstr1`, contains one occurrence of each tag. Since the expression uses `()?` around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, `hstr2`, does not contain the code tags at all. Just the same, the expression matches because `()?` allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr2, expr, 'match')
ans =
    '</a>%%<br>'
```

Zero or More — expr*

The first `regexp` command looks for at least one occurrence of `<br>` and finds it. The second command parses a different string for at least one `<br>` and fails. The third command uses `*` to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*(<br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';

hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*(<br>).*</p>', 'match')
ans =
     {}

regexp(hstr2, '<p>.*(<br>)*.*</p>', 'match')
```

```
ans =
    '<p>This string has no line breaks</p>';
```

One or More — expr+

Use + to verify that the HTML image source is not empty. This looks for one or more characters in the gif filename:

```
hstr = '<a href="s12.html"><img src="b_prev.gif" border=0>';
expr = '<img src="\w+.gif';

regexp(hstr, expr, 'match')
ans =
    '<img src="b_prev.gif'
```

Exact, Minimum, and Maximum Quantities — {min,max}

Use {m}, {m,}, and {m,n} to verify the href syntax used in HTML. This statement requires the href to have at least one non-whitespace character, followed by exactly one occurrence of .html, optionally followed by # and five to eight digits:

```
hstr = '<a name="18749"></a><a href="s13.html#18760">';
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}"';

regexp(hstr, expr, 'match')
ans =
    '<a href="s13.html#18760"'
```

Lazy Quantifiers — expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (?). The first part of the example uses the default quantifier to match all characters from the opening <tr to the ending </td:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';
regexp(hstr, '</?t.*>', 'match')
ans =
```

```
      '<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens*. You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

- "Operators Used with Tokens" on page 3-48
- "Introduction to Using Tokens" on page 3-49
- "Using Tokens — Example 1" on page 3-50
- "Using Tokens — Example 2" on page 3-50
- "Tokens That Are Not Matched" on page 3-51
- "Using Tokens in a Replacement String" on page 3-53

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.
`\N`	Match the Nth token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.

Operator	Usage
`$N`	Insert the match for the Nth token in the replacement string. Used only by the `regexprep` function. If `N` is equal to zero, then insert the entire match in the replacement string.
`(?(N)s1|s2)`	If N^th token is found, then match `s1`, else match `s2`

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use '`(\$\d+)`'. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use `\3`.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown below, the `(\S)` phrase creates a token whenever `regexp` matches any non-whitespace character in the string. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
   'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array `tok` are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string `poestr` are:

```
11 11,  26 26,  35 35,  57 57
```

Using Tokens — Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern `and(y|rew)` finds the text `andrew`, token 1 is assigned the value `rew`. However, if the search pattern `(and(y|rew))` is used, token 1 is assigned the value `andrew`.

Using Tokens — Example 2

Use `(expr)` and `\N` to capture pairs of matching HTML tags (e.g., `<a>` and `<\a>`) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, '<(\w+)', matches an opening bracket (<) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, '.*?>.*?', matches the remainder of this HTML tag (characters up to the >), and any characters that may precede the next opening bracket.

The last part, '</\1>', matches all characters in the ending HTML tag. This tag is composed of the sequence </tag>, where tag is whatever characters were captured as a token.

```
hstr = '<!comment><a name="752507"></a><b>Default</b><br>';
expr = '<(\w+).*?>.*?</\1>';

[mat tok] = regexp(hstr, expr, 'match', 'tokens');
mat{:}
ans =
    <a name="752507"></a>
ans =
    <b>Default</b>

tok{:}
ans =
    'a'
ans =
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, regexp and regexpi return an empty string ('') as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes regexp on the path string str returned from the MATLAB tempdir function. The regular expression expr includes six token specifiers, one for each piece of the path string. The third specifier

[a-z]+ has no match in the string because this part of the path, Profiles, begins with an uppercase letter:

```
str = tempdir
str =
   C:\WINNT\Profiles\bpascal\LOCALS~1\Temp\

expr = ['([A-Z]:)\\(WINNT)\\([a-z]+)?.*\\' ...
          '([a-z]+)\\([A-Z]+~\d)\\(Temp)\\'];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (''). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
ans =
    'C:'    'WINNT'     ''    'bpascal'    'LOCALS~1'    'Temp'
```

The third token extent returned in the variable ext has the starting index set to 10, which is where the nonmatching substring, Profiles, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using $1, $2, etc. instead of \1, \2, etc. This example captures two tokens and reverses their order. The first, $1, is 'Norma Jean' and the second, $2, is 'Baker'. Note that regexprep returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.
\k<name>	Match the token referred to by name.
$<name>	Insert the match for named token in a replacement string. Used only with the regexprep function.
(?(name)s1|s2)	If named token is found, then match s1; otherwise, match s2

When referencing a named token within the expression, use the syntax \k<name> instead of the numeric \1, \2, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

regexp(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
     adrs: '134 Main Street'
     city: 'Boulder'
    state: 'CO'
      zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
     adrs: '26 Walnut Road'
     city: 'Topeka'
    state: 'KA'
      zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
     adrs: '847 Industrial Drive'
     city: 'Elizabeth'
```

```
    state: 'NJ'
      zip: '73548'
```

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an `if-then` or an `if-then-else` clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`
`(?(cond)`$expr_1$`|`$expr_2$`)`	If condition `cond` is true, then match expression $expr_1$. Otherwise match expression $expr_2$

The first entry in this table is the same as an `if-then` statement. MATLAB tests the state of condition `cond` and then matches expression `expr` only if the condition was found to be true. In the form of an `if-then` statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an `if-then-else` statement. If the condition is true, MATLAB matches $expr_1$; if false, it matches $expr_2$ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition `cond` in either of these syntaxes can be any one of the following:

- A specific token, identified by either number or name, is located in the input string. See "Conditions Based on Tokens" on page 3-56, below.
- A lookaround operation results in a match. See "Conditions Based on a Lookaround Match" on page 3-57, below.
- A dynamic expression of the form `(?@cmd)` returns a nonzero numeric value. See "Conditions Based on Return Values" on page 3-57, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as `1`, `2`, and `3`.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, "**if** token `1` is found (i.e., `Mr` is followed by the letter `s`), **then** match `her`, **else** match `his`."

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';

[mat tok] = regexp('Mr. Clark went to see his son', ...
   expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
ans =
     ''    'his'
```

In the second part of the example, the token `s` is found and MATLAB matches the word `her`:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}
```

```
tok{:}
ans =
    's'    'her'
```

Note When referring to a token within a condition, use just the number of the token. For example, refer to token 2 by using the number 2 alone, and not \2 or $2.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the `else` part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form `(?@cmd)`, enables you to execute a MATLAB command (shown here as `cmd`) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want `regexp` to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the `regexprep` function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the `match_expr` or `replace_expr` arguments of these commands:

```
regexp(string, match_expr)
regexpi(string, match_expr)
regexprep(string, match_expr, replace_expr)
```

MATLAB supports three types of dynamic operators for use in a match expression. See “Dynamic Operators for the Match Expression” on page 3-59 for more information.

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(?@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the match expression. This is a combination of the two dynamic syntaxes shown above: `(??expr)` and `(?@cmd)`.

MATLAB supports one type of dynamic expression for use in the replacement expression of a `regexprep` command. See “Dynamic Operators for the Replacement Expression” on page 3-64 for more information.

Operator	Usage
`${cmd}`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following `regexprep` command correctly replaces the term `internationalization` with its abbreviated form, `i18n`. However, to use it on a different term such as `globalization`, you have to use a different replacement expression:

```
match_expr = '(^\w)(\w*)(\w$)';
```

```
replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n

replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the replacement expression on the input string so that you do not have to change the expression each time. This example uses the dynamic syntax `${cmd}` from the second table shown above:

```
match_expr = '(^\w)(\w*)(\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- "Dynamic Expressions that Modify the Match Expression — (??expr)" on page 3-60

  "Dynamic Commands that Modify the Match Expression — (??@cmd)" on page 3-61

  "Dynamic Commands that Serve a Functional Purpose — (?@cmd)" on page 3-62

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When MATLAB

evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions that Modify the Match Expression — (??expr). The `(??expr)` operator parses expression `expr`, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of `X` characters in each of the strings stored in the input cell array. Note however that the number of `X`s varies in each string. If the count did not vary, you could use the expression `X{n}` to indicate that you want to match `n` of these characters. But, a constant value of `n` does not work in this case.

The solution used here is to capture the leading count number (e.g., the `5` in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is `(??X{$1})`, where `$1` is the value captured by the token `\d+`. The metacharacter `{$1}` makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell array. With the first input string, `regexp` looks for five `X` characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands that Modify the Match Expression — (??@cmd). MATLAB uses the `(??@function)` operator to include the results of a MATLAB command in the match expression. This command must return a string that can be used within the match expression.

The `regexp` command below uses the dynamic expression `(??@flilplr($1))` to locate a palindrome string, "Never Odd or Even", that has been embedded into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the string, and then attempts to match as much of the reversed-order string as possible. This requires a dynamic expression because the value for `$1` relies on the value of the token `(.{3,})`:

```
% Put the string in lowercase.
str = lower(...
  'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
   findthepalindromeneveroddoreveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
   'neveroddoreven'
```

Dynamic expressions in MATLAB have access to the currently active workspace. This means that you can change any of the functions or variables used in a dynamic expression just by changing variables in the workspace. Repeat the last command of the example above, but this time define the function to be called within the expression using a function handle stored in the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
```

```
    'neveroddoreven'
```

Dynamic Commands that Serve a Functional Purpose — (?@cmd). The `(?@cmd)` operator specifies a MATLAB command that `regexp` or `regexprep` is to run while parsing the overall match expression. Unlike the other dynamic expressions in MATLAB, this operator does not alter the contents of the expression it is used in. Instead, you can use this functionality to get MATLAB to report just what steps it's taking as it parses the contents of one of your regular expressions.

The following example parses a word for zero or more characters followed by two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the example inserts a short script `(?@disp($1))` in the expression to display the characters that finally constitute the match. Because the example uses greedy quantifiers, MATLAB attempts to match as much of the string as possible. So, even though MATLAB finds a match toward the beginning of the string, it continues to look for more matches until it arrives at the very end of the string. From there, it backs up through the letters `i` then `p` and the next `p`, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy (`*?`). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*?(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has matched the string quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*?(\w)(?@disp($1))\1\w*';)
m
i
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The `(?!)` metacharacter found at the end of the expression is actually an empty lookahead operator, and forces a failure at each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script `(?@if(~isempty($&)))` serves to omit these strings from the `matches` cell array:

```
matches = {};
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)),' ...
   'matches{end+1}=$&;end)(?!)'];

regexp('Euler Cauchy Boole', expr);

matches
matches =
    'Euler Cauchy Boole'    'Euler Cauchy '    'Euler '
'Cauchy Boole'    'Cauchy '    'Boole'
```

The metacharacters `$&` (or the equivalent `$0`), `` $` ``, and `$'` refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These metacharacters are sometimes useful when working with dynamic expressions, particularly those that employ the `(?@cmd)` operator.

This example parses the input string looking for the letter `g`. At each iteration through the string, `regexp` compares the current character with `g`, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a `^` character.

(The $` and $· metacharacters capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks ($'') to express the sequence $· when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf(''starting match: [%s^%s]'',$`,$'')))g';

regexp(str, expr, 'once');
starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the `regexprep` function.

Dynamic Commands that Modify the Replacement Expression — ${cmd}. The `${cmd}` operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the `regexprep` call shown here, the replacement string is `'${convert($1,$2)}'`. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute an M-file function named `convert` using the two tokens `(\d+\.?\d*)` and `(\w+)`, derived from the

string being matched, as input arguments in the call to `convert`. The replacement string requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the M-file named `convert` that converts measurements from imperial units to metric. To convert values from the string being parsed, `regexprep` calls the `convert` function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout  = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093;  uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536;  uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731;  uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;   uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];


regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This highway is 201.1625 kilometers long


regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This pitcher holds 1.1828 litres of water


regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

```
ans =
   This stone weighs about 4.536 kilograms
```

As with the `(??@ )` operator discussed in an earlier section, the `${ }` operator has access to variables in the currently active workspace. The following `regexprep` command uses the array `A` defined in the base workspace:

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf(''%d%d%d '', A)}'})
ans =
The columns of matrix A are 834 159 672
```

String Replacement

The `regexprep` function enables you to replace a string that is identified by a regular expression with another string. The following syntax replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of the metacharacters shown in the following table:

Operator	Usage
Operators from Character Representation on page 3-73 table	The character represented by the metacharacter sequence
`` $` ``	That part of the input string that precedes the current match

Operator	Usage
`$&` or `$0`	That part of the input string that is currently a match
`$·`	That part of the input string that follows the current match. In MATLAB, use `$''` to represent the character sequence `$·`.
`$N`	The string represented by the token identified by `name`
`$<name>`	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator `(...)`. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See the section on "Tokens" on page 3-48 and the example "Using Tokens in a Replacement String" on page 3-53 in this documentation for information on using tokens.)

Note When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token `2` by using `$2`, and not `2` or `\2`.

The following example uses both the `${cmd}` and `$N` operators in the replacement strings of nested `regexprep` commands to capitalize the first letter of each sentence. The inner `regexprep` looks for the start of the entire string and capitalizes the single instance; the outer `regexprep` looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
```

```
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '(^.)', '${upper($1)}'), ...
   '(?<=\.\s*)([a-z])','${upper($1)}')

ans =
Here are a few sentences. None are capitalized. Let's change that.
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

This section covers the following topics:

- "Finding a Single Pattern in Multiple Strings" on page 3-68
- "Finding Multiple Patterns in Multiple Strings" on page 3-70
- "Replacing Multiple Strings" on page 3-70

Finding a Single Pattern in Multiple Strings

The example shown here uses the `regexp` function on a cell array of strings `cstr`. It searches each string of the cell array for consecutive matching letters (e.g., `'oo'`). The function returns a cell array of the same size as the input array. Each row of the return array contains the indices for which there was a match against the input cell array.

Here is the input cell array:

```
cstr = {                                  ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

Find consecutive matching letters by capturing a letter as a token `(.)` and then repeating that letter as a token reference, `\1`:

```
idx = regexp(cstr, '(.)\1');

whos idx
  Name      Size                    Bytes  Class

  idx       4x1                       296  cell array

idx{:}
ans =                % 'Whose woods these are I think I know.'
    8                %          |8

ans =                % 'His house is in the village though;'
   23                %                         |23

ans =                % 'He will not see me stopping here'
    6    14    23    %        |6      |14      |23

ans =                % 'To watch his woods fill up with snow.'
   15    22          %                 |15    |22
```

To return substrings instead of indices, use the `'match'` parameter:

```
mat = regexp(cstr, '(.)\1', 'match');
mat{3}
ans =
   'll'    'ee'    'pp'
```

Finding Multiple Patterns in Multiple Strings

This example uses a cell array of strings in both the input string and the expression. The two cell arrays are of different shapes: `cstr` is 4-by-1 while `expr` is 1-by-4. The command is valid as long as they both have the same number of cells.

Find uppercase or lowercase `'i'` followed by a white-space character in `str{1}`, the sequence `'hou'` in `str{2}`, two consecutive matching letters in `str{3}`, and words beginning with `'w'` followed by a vowel in `str{4}`.

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
idx = regexpi(cstr, expr);

idx{:}
ans =                  % 'Whose woods these are I think I know.'
   23    31            %                          |23     |31

ans =                  % 'His house is in the village though;'
    5    30            %      |5                         |30

ans =                  % 'He will not see me stopping here'
    6    14    23      %       |6      |14       |23

ans =                  % 'To watch his woods fill up with snow.'
    4    14    28      %     |4        |14            |28
```

Note that the returned cell array has the dimensions of the input string, `cstr`. The dimensions of the return value are always derived from the input string, whenever the input string is a cell array. If the input string is not a cell array, then it is the dimensions of the expression that determine the shape of the return array.

Replacing Multiple Strings

When replacing multiple strings with `regexprep`, use a single replacement string if the expression consists of a single string. This example uses a common replacement value (`'--'`) for all matches found in the multiple string input `cstr`. The function returns a cell array of strings having the same dimensions as the input cell array:

```
s = regexprep(cstr, '(.)\1', '--', 'ignorecase')
```

```
s =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

You can use multiple replacement strings if the expression consists of multiple strings. In this example, the input string and replacement string are both 4-by-1 cell arrays, and the expression is a 1-by-4 cell array. As long as the expression and replacement arrays contain the same number of elements, the statement is valid. The dimensions of the return value match the dimensions of the input string:

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
repl = {'-1-'; '-2-'; '-3-'; '-4-'};

s = regexprep(cstr, expr, repl, 'ignorecase')
s =
    'Whose w-3-ds these are -1-think -1-know.'
    'His -2-se is in the vi-3-age t-2-gh;'
    'He -4--3- not s-3- me sto-3-ing here'
    'To -4-tch his w-3-ds fi-3- up -4-th snow.'
```

Operator Summary

MATLAB provides these operators for working with regular expressions:

- Character Classes on page 3-72
- Character Representation on page 3-73
- “Grouping Operators” on page 3-37
- “Nonmatching Operators” on page 3-39
- “Positional Operators” on page 3-39
- Lookaround Operators on page 3-74
- Quantifiers on page 3-75
- Ordinal Token Operators on page 3-76
- Named Token Operators on page 3-76

- Conditional Expression Operators on page 3-77
- Dynamic Expression Operators on page 3-77
- Replacement String Operators on page 3-78

Character Classes

Operator	Usage
.	Any single character, including white space
$[c_1c_2c_3]$	Any character contained within the brackets: c_1 or c_2 or c_3
$[\hat{}c_1c_2c_3]$	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
$[c_1\text{-}c_2]$	Any character in the range of c_1 through c_2
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any non-whitespace character; equivalent to [^ \f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character; equivalent to [a-zA-Z_0-9]. (True only for English character sets).
\W	Any character that is not alphabetic, numeric, or underscore; equivalent to [^a-zA-Z_0-9]. (True only for English character sets).
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]

Character Classes (Continued)

Operator	Usage
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`

Character Representation

Operator	Usage
`\\`	Backslash
`\$`	Dollar sign
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Grouping Operators

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.

Grouping Operators (Continued)

Operator	Usage
(?>expr)	Group atomically.
expr$_1$|expr$_2$	Match expression expr$_1$ or expression expr$_2$.

Nonmatching Operators

Operator	Usage
(?#comment)	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
^expr	Match expr if it occurs at the beginning of the input string.
expr$	Match expr if it occurs at the end of the input string.
\<expr	Match expr when it occurs at the beginning of a word.
expr\>	Match expr when it occurs at the end of a word.
\<expr\>	Match expr when it represents the entire word.

Lookaround Operators

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found

Lookaround Operators (Continued)

Operator	Usage
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Quantifiers

Operator	Usage
`expr{m,n}`	Match `expr` when it occurs at least `m` times but no more than `n` times consecutively.
`expr{m,}`	Match `expr` when it occurs at least `m` times consecutively.
`expr{n}`	Match `expr` when it occurs exactly `n` times consecutively. Equivalent to `{n,n}`.
`expr?`	Match `expr` when it occurs 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match `expr` when it occurs 0 or more times consecutively. Equivalent to `{0,}`.
`expr+`	Match `expr` when it occurs 1 or more times consecutively. Equivalent to `{1,}`.
`q_expr*`	Match as much of the quantified expression as possible, where `q_expr` represents any of the expressions shown in the first six rows of this table.

Quantifiers (Continued)

Operator	Usage
q_expr+	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
q_expr?	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
(expr)	Capture in a token all characters matched by the expression within the parentheses.
\N	Match the Nth token generated by this command. That is, use \1 to match the first token, \2 to match the second, and so on.
$N	Insert the match for the Nth token in the replacement string. Used only by the regexprep function. If N is equal to zero, then insert the entire match in the replacement string.
(?(N)s1|s2)	If N^{th} token is found, then match s1, else match s2

Named Token Operators

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.
\k<name>	Match the token referred to by name.

Named Token Operators (Continued)

Operator	Usage
`$<name>`	Insert the match for named token in a replacement string. Used only with the `regexprep` function.
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`

Conditional Expression Operators

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`
(?(cond)$expr_1$|$expr_2$)	If condition `cond` is true, then match expression $expr_1$. Otherwise match expression $expr_2$

Dynamic Expression Operators

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.

Dynamic Expression Operators (Continued)

Operator	Usage
(?@cmd)	Execute the MATLAB command cmd, and include the string returned by cmd in the match expression. This is a combination of the two dynamic syntaxes shown above: (??expr) and (?@cmd).
${cmd}	Execute the MATLAB command cmd, and include the string returned by cmd in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from Character Representation on page 3-73 table	The character represented by the metacharacter sequence
$`	That part of the input string that precedes the current match
$& or $0	That part of the input string that is currently a match
$·	That part of the input string that follows the current match. In MATLAB, use $'' to represent the character sequence $·.
$N	The string represented by the token identified by name
$<name>	The string represented by the token identified by name
${cmd}	The string returned when MATLAB executes the command cmd

Comma-Separated Lists

In this section...

"What Is a Comma-Separated List?" on page 3-79

"Generating a Comma-Separated List" on page 3-79

"Assigning Output from a Comma-Separated List" on page 3-81

"Assigning to a Comma-Separated List" on page 3-82

"How to Use the Comma-Separated Lists" on page 3-83

"Fast Fourier Transform Example" on page 3-85

What Is a Comma-Separated List?

Typing in a series of numbers separated by commas gives you what is called a *comma-separated list*. MATLAB returns each value individually:

```
1, 2, 3
ans =
     1
ans =
     2
ans =
     3
```

Such a list, by itself, is not very useful. But when used with large and more complex data structures like MATLAB structures and cell arrays, the comma-separated list can enable you to simplify your MATLAB code.

Generating a Comma-Separated List

This section describes how to generate a comma-separated list from either a cell array or a MATLAB structure.

Generating a List from a Cell Array

Extracting multiple elements from a cell array yields a comma-separated list. Given a 4-by-6 cell array as shown here

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

C
C =
    [2]    [10]    [18]    [26]    [34]    [42]
    [4]    [12]    [20]    [28]    [36]    [44]
    [6]    [14]    [22]    [30]    [38]    [46]
    [8]    [16]    [24]    [32]    [40]    [48]
```

extracting the fifth column generates the following comma-separated list:

```
C{:, 5}
ans =
    34
ans =
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
C{1, 5}, C{2, 5}, C{3, 5}, C{4, 5}
```

Generating a List from a Structure

For structures, extracting a field of the structure that exists across one of its dimensions yields a comma-separated list.

Start by converting the cell array used above into a 4-by-1 MATLAB structure with six fields: `f1` through `f6`. Read field `f5` for all rows and MATLAB returns a comma-separated list:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

S.f5
ans =
    34
ans =
```

```
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
S(1).f5, S(2).f5, S(3).f5, S(4).f5
```

Assigning Output from a Comma-Separated List

You can assign any or all consecutive elements of a comma-separated list to variables with a simple assignment statement. Using the cell array `C` from the previous section, assign the first row to variables `c1` through `c6`:

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

[c1 c2 c3 c4 c5 c6] = C{1,1:6};

c5
c5 =
    34
```

If you specify fewer output variables than the number of outputs returned by the expression, MATLAB assigns the first N outputs to those N variables, and then discards any remaining outputs. In this next example, MATLAB assigns `C{1,1:3}` to the variables `c1`, `c2`, and `c3`, and then discards `C{1,4:6}`:

```
[c1 c2 c3] = C{1,1:6};
```

You can assign structure outputs in the same manner:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

[sf1 sf2 sf3] = S.f5;

sf3
sf3 =
    38
```

You also can use the deal function for this purpose.

Assigning to a Comma-Separated List

The simplest way to assign multiple values to a comma-separated list is to use the deal function. This function distributes all of its input arguments to the elements of a comma-separated list.

This example initializes a comma-separated list to a set of vectors in a cell array, and then uses deal to overwrite each element in the list:

```
c{1} = [31 07];    c{2} = [03 78];

c{:}
ans =
    31     7
ans =
     3    78


[c{:}] = deal([10 20],[14 12]);

c{:}
ans =
    10    20
ans =
    14    12
```

This example does the same as the one above, but with a comma-separated list of vectors in a structure field:

```
s(1).field1 = [31 07];    s(2).field1 = [03 78];

s.field1
ans =
    31     7
ans =
     3    78
```

```
[s.field1] = deal([10 20],[14 12]);

s.field1
ans =
    10    20
ans =
    14    12
```

How to Use the Comma-Separated Lists

Common uses for comma-separated lists are

- "Constructing Arrays" on page 3-83
- "Displaying Arrays" on page 3-84
- "Concatenation" on page 3-84
- "Function Call Arguments" on page 3-84
- "Function Return Values" on page 3-85

The following sections provide examples of using comma-separated lists with cell arrays. Each of these examples applies to MATLAB structures as well.

Constructing Arrays

You can use a comma-separated list to enter a series of elements when constructing a matrix or array. Note what happens when you insert a *list* of elements as opposed to adding the cell itself.

When you specify a list of elements with `C{:, 5}`, MATLAB inserts the four individual elements:

```
A = {'Hello', C{:, 5}, magic(4)}
A =
    'Hello'    [34]    [36]    [38]    [40]    [4x4 double]
```

When you specify the `C` cell itself, MATLAB inserts the entire cell array:

```
A = {'Hello', C, magic(4)}
A =
    'Hello'    {4x6 cell}    [4x4 double]
```

Displaying Arrays

Use a list to display all or part of a structure or cell array:

```
A{:}
ans =
    Hello
ans =
    34
ans =
    36
ans =
    38
  .
  .
  .
```

Concatenation

Putting a comma-separated list inside square brackets extracts the specified elements from the list and concatenates them:

```
A = [C{:, 5:6}]
A =
    34    36    38    40    42    44    46    48

whos A
  Name      Size                       Bytes  Class

  A         1x8                           64  double array
```

Function Call Arguments

When writing the code for a function call, you enter the input arguments as a list with each argument separated by a comma. If you have these arguments stored in a structure or cell array, then you can generate all or part of the argument list from the structure or cell array instead. This can be especially useful when passing in variable numbers of arguments.

This example passes several attribute-value arguments to the plot function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';         C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';   C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Function Return Values

MATLAB functions can also return more than one value to the caller. These values are returned in a list with each value separated by a comma. Instead of listing each return value, you can use a comma-separated list with a structure or cell array. This becomes more useful for those functions that have variable numbers of return values.

This example returns four values to a cell array:

```
C = cell(1, 4);
[C{:}] = fileparts('work/mytests/strArrays.mat')
C =
    'work/mytests'    'strArrays'    '.mat'     ''
```

Fast Fourier Transform Example

The fftshift function swaps the left and right halves of each dimension of an array. For a simple vector such as [0 2 4 6 8 10] the output would be [6 8 10 0 2 4]. For a multidimensional array, fftshift performs this swap along each dimension.

fftshift uses vectors of indices to perform the swap. For the vector shown above, the index [1 2 3 4 5 6] is rearranged to form a new index [4 5 6 1 2 3]. The function then uses this index vector to reposition the elements. For a multidimensional array, fftshift must construct an index vector for each dimension. A comma-separated list makes this task much simpler.

Here is the fftshift function:

```
function y = fftshift(x)
```

```
numDims = ndims(x);
idx = cell(1, numDims);

for k = 1:numDims
   m = size(x, k);
   p = ceil(m/2);
   idx{k} = [p+1:m 1:p];
   end

y = x(idx{:});
```

The function stores the index vectors in cell array `idx`. Building this cell array is relatively simple. For each of the `N` dimensions, determine the size of that dimension and find the integer index nearest the midpoint. Then, construct a vector that swaps the two halves of that dimension.

By using a cell array to store the index vectors and a comma-separated list for the indexing operation, `fftshift` shifts arrays of any dimension using just a single operation: `y = x(idx{:})`. If you were to use explicit indexing, you would need to write one `if` statement for each dimension you want the function to handle:

```
if ndims(x) == 1
   y = x(index1);
else if ndims(x) == 2
   y = x(index1, index2);
end
```

Another way to handle this without a comma-separated list would be to loop over each dimension, converting one dimension at a time and moving data each time. With a comma-separated list, you move the data just once. A comma-separated list makes it very easy to generalize the swapping operation to an arbitrary number of dimensions.

Program Control Statements

In this section...

"Conditional Control — if, switch" on page 3-87

"Loop Control — for, while, continue, break" on page 3-91

"Error Control — try, catch" on page 3-94

"Program Termination — return" on page 3-95

Conditional Control — if, switch

This group of control statements enables you to select at run-time which block of code is executed. To make this selection based on whether a condition is true or false, use the `if` statement (which may include `else` or `elseif`). To select from a number of possible options depending on the value of an expression, use the `switch` and `case` statements (which may include `otherwise`).

if, else, and elseif

`if` evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is `true` (that is, if it evaluates to logical `1`), MATLAB executes all the statements between the `if` and `end` lines. It resumes execution at the line following the `end` statement. If the condition is `false` (evaluates to logical `0`), MATLAB skips all the statements between the `if` and `end` lines, and resumes execution at the line following the `end` statement.

For example,

```
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

You can nest any number of `if` statements.

If the logical expression evaluates to a nonscalar value, all the elements of the argument must be nonzero. For example, assume `X` is a matrix. Then the statement

```
if X
    statements
end
```

is equivalent to

```
if all(X(:))
    statements
end
```

The `else` and `elseif` statements further conditionalize the `if` statement:

- The `else` statement has no logical condition. The statements associated with it execute if the preceding `if` (and possibly `elseif` condition) evaluates to logical `0` (`false`).
- The `elseif` statement has a logical condition that it evaluates if the preceding `if` (and possibly `elseif` condition) is `false`. The statements associated with it execute if its logical condition evaluates to logical `1` (`true`). You can have multiple `elseif` statements within an `if` block.

```
if n < 0             % If n negative, display error message.
    disp('Input must be positive');
elseif rem(n,2) == 0 % If n positive and even, divide by 2.
    A = n/2;
else
    A = (n+1)/2;     % If n positive and odd, increment and divide.
end
```

if Statements and Empty Arrays. An `if` condition that reduces to an empty array represents a `false` condition. That is,

```
if A
   S1
else
   S0
```

```
end
```

executes statement S0 when A is an empty array.

switch, case, and otherwise

switch executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)
   case value1
      statements       % Executes if expression is value1
   case value2
      statements       % Executes if expression is value2
    .
    .
    .
   otherwise
      statements       % Executes if expression does not
                        % match any case
end
```

This block consists of

- The word switch followed by an expression to evaluate.
- Any number of case groups. These groups consist of the word case followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another switch block. Execution of a case group ends when MATLAB encounters the next case statement or the otherwise statement. Only the first matching case is executed.
- An optional otherwise group. This consists of the word otherwise, followed by the statements to execute if the expression's value is not handled by any of the preceding case groups. Execution of the otherwise group ends at the end statement.
- An end statement.

switch works by comparing the input expression to each case value. For numeric expressions, a case statement is true if (value==expression). For string expressions, a case statement is true if strcmp(value,expression).

The code below shows a simple example of the switch statement. It checks the variable input_num for certain values. If input_num is -1, 0, or 1, the case statements display the value as text. If input_num is none of these values, execution drops to the otherwise statement and the code displays the text 'other value'.

```
switch input_num
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
         disp('other value');
end
```

Note For C programmers, unlike the C language switch construct, the MATLAB switch does not “fall through.” That is, if the first case statement is true, other case statements do not execute. Therefore, break statements are not used.

---

switch can handle multiple conditions in a single case statement by enclosing the case expression in a cell array.

```
switch var
    case 1
        disp('1')
    case {2,3,4}
        disp('2 or 3 or 4')
    case 5
        disp('5')
    otherwise
        disp('something else')
end
```

Loop Control — for, while, continue, break

With loop control statements, you can repeatedly execute a block of code, looping back through the block while keeping track of each iteration with an incrementing index variable. Use the `for` statement to loop a specific number of times. The `while` statement is more suitable for basing the loop execution on how long a condition continues to be true or false. The `continue` and `break` statements give you more control on exiting the loop.

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Techniques for Improving Performance" on page 11-4 for more information on this.

---

for

The `for` loop executes a statement or group of statements a predetermined number of times. Its syntax is

```
for index = start:increment:end
    statements
end
```

The default increment is `1`. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the *end* value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for n = 2:6
    x(n) = 2 * x(n - 1);
end
```

You can nest multiple `for` loops.

```
for m = 1:5
    for n = 1:100
        A(m, n) = 1/(m + n - 1);
    end
```

```
end
```

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Vectorizing Loops" on page 11-4 for details.

Using Arrays as Indices. The index of a `for` loop can be an array. For example, consider an m-by-n array `A`. The statement

```
for k = A
    statements
end
```

sets `k` equal to the vector `A(:,i)`, where `i` is the iteration number of the loop. For the first loop iteration, `k` is equal to `A(:,1)`; for the second, `k` is equal to `A(:,2)`; and so on until `k` equals `A(:,n)`. That is, the loop iterates for a number of times equal to the number of columns in `A`. For each iteration, `k` is a vector containing one of the columns of `A`.

while

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is true (`1`). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be `1` for execution to continue. To reduce a matrix to a scalar value, use the `all` and `any` functions.

For example, this `while` loop finds the first integer `n` for which `n!` (`n` factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a `while` loop at any time using the `break` statement.

while Statements and Empty Arrays. A `while` condition that reduces to an empty array represents a `false` condition. That is,

```
while A, S1, end
```

never executes statement `S1` when `A` is an empty array.

continue

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file, `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m', 'r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line, '%', 1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines', count));
```

break

The `break` statement terminates the execution of a `for` loop or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside of the loop. In nested loops, `break` exits from the innermost loop only.

The example below shows a while loop that reads the contents of the file fft.m into a MATLAB character array. A break statement is used to exit the while loop when the first empty line is encountered. The resulting character array contains the M-file help for the fft program.

```
fid = fopen('fft.m', 'r');
s = '';
while ~feof(fid)
   line = fgetl(fid);
   if isempty(line)
      break
   end
   s = strvcat(s, line);
end
disp(s)
```

Error Control — try, catch

Error control statements provide a way for you to take certain actions in the event of an error. Use the try statement to test whether a certain command in your code generates an error. If an error does occur within the try block, MATLAB immediately jumps to the corresponding catch block. The catch part of the statement needs to respond in some way to the error.

try and catch

The general form of a try-catch statement sequence is

```
try
   statement
   ...
   statement
catch
   statement
   ...
   statement
end
```

In this sequence, the statements between try and catch are executed until an error occurs. The statements between catch and end are then executed. Use lasterr to see the cause of the error. If an error occurs between catch

and `end`, MATLAB terminates execution unless another `try-catch` sequence has been established.

Program Termination — return

Program termination control enables you to exit from your program at some point prior to its normal termination point.

return

After a MATLAB function runs to completion, it terminates and returns control either to the function that called it, or to the keyboard. If you need to exit a function prior to the point of normal completion, you can force an early termination using the `return` function. `return` immediately terminates the current sequence of commands and exits the currently running function.

`return` is also used to terminate `keyboard` mode.

Symbol Reference

In this section...

"Asterisk — *" on page 3-96
"At — @" on page 3-97
"Colon — :" on page 3-98
"Comma — ," on page 3-99
"Curly Braces — { }" on page 3-100
"Dot — ." on page 3-100
"Dot-Dot — .." on page 3-101
"Dot-Dot-Dot (Ellipsis) — ..." on page 3-101
"Dot-Parentheses — .( )" on page 3-102
"Exclamation Point — !" on page 3-103
"Parentheses — ( )" on page 3-103
"Percent — %" on page 3-103
"Percent-Brace — %{ %}" on page 3-104
"Semicolon — ;" on page 3-104
"Single Quotes — ' '" on page 3-105
"Space Character" on page 3-106
"Slash and Backslash — / \" on page 3-106
"Square Brackets — [ ]" on page 3-107

This section does not include symbols used in arithmetic, relational, and logical operations. For a description of these symbols, see the top of the list. "Functions — Alphabetical List" in the MATLAB Help browser.

Asterisk — *

An asterisk in a filename specification is used as a wildcard specifier, as described below.

Filename Wildcard

Wildcards are generally used in file operations that act on multiple files or directories. They usually appear in the string containing the file or directory specification. MATLAB matches all characters in the name exactly except for the wildcard character `*`, which can match any one or more characters.

To locate all files with names that start with `'january_'` and have a `mat` file extension, use

```
dir('january_*.mat')
```

You can also use wildcards with the `who` and `whos` functions. To get information on all variables with names starting with `'image'` and ending with `'Offset'`, use

```
whos image*Offset
```

At — @

The `@` sign signifies either a function handle constructor or a directory that supports a MATLAB class.

Function Handle Constructor

The `@` operator forms a handle to either the named function that follows the `@` sign, or to the anonymous function that follows the `@` sign.

Function Handles in General. Function handles are commonly used in passing functions as arguments to other functions. Construct a function handle by preceding the function name with an `@` sign:

```
fhandle = @myfun
```

You can read more about function handles in “Function Handles” on page 4-22.

Handles to Anonymous Functions. Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function and a handle to that function using the syntax

```
fhandle = @(arglist) body
```

where body defines the body of the function and arglist is the list of arguments you can pass to the function.

See "Anonymous Functions" on page 5-3 for more information.

Class Directory Designator

A MATLAB class directory contains source files that define the methods and properties of a class. All MATLAB class directory names must begin with an @ sign:

```
\@myclass\get.m
```

See "MATLAB Classes" on page 2-117 for more information.

Colon — :

The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays. See "Generating a Numeric Sequence" on page 1-11 for more information on using the colon operator.

Numeric Sequence Range

Generate a sequential series of regularly spaced numbers from first to last using the syntax first:last. For an incremental sequence from 6 to 17, use

```
N = 6:17
```

Numeric Sequence Step

Generate a sequential series of numbers, each number separated by a step value, using the syntax first:step:last. For a sequence from 2 through 38, stepping by 4 between each entry, use

```
N = 2:4:38
```

Indexing Range Specifier

Index into multiple rows or columns of a matrix using the colon operator to specify a range of indices:

```
B = A(7, 1:5);          % Read columns 1-5 of row 7.
B = A(4:2:8, 1:5);      % Read columns 1-5 of rows 4, 6, and 8.
B = A(:, 1:5);          % Read columns 1-5 of all rows.
```

Conversion to Column Vector

Convert a matrix or array to a column vector using the colon operator as a single index:

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

Preserving Array Shape on Assignment

Using the colon operator on the left side of an assignment statement, you can assign new values to array elements without changing the shape of the array:

```
A = rand(3,4);
A(:) = 1:12;
```

Comma — ,

A comma is used to separate the following types of elements.

Row Element Separator

When constructing an array, use a comma to separate elements that belong in the same row:

```
A = [5.92, 8.13, 3.53]
```

Array Index Separator

When indexing into an array, use a comma to separate the indices into each dimension:

```
X = A(2, 7, 4)
```

Function Input and Output Separator

When calling a function, use a comma to separate output and input arguments:

```
function [data, text] = xlsread(file, sheet, range, mode)
```

Command or Statement Separator

To enter more than one MATLAB command or statement on the same line, separate each command or statement with a comma:

```
for k = 1:10,   sum(A(k)),   end
```

Curly Braces — { }

Use curly braces to construct or get the contents of cell arrays.

Cell Array Constructor

To construct a cell array, enclose all elements of the array in curly braces:

```
C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}
```

Cell Array Indexing

Index to a specific cell array element by enclosing all indices in curly braces:

```
A = C{4,7,2}
```

See "Cell Arrays" on page 2-93 for more information.

Dot — .

The single dot operator has the following different uses in MATLAB.

Structure Field Definition

Add fields to a MATLAB structure by following the structure name with a dot and then a field name:

```
funds(5,2).bondtype = 'Corporate';
```

See "Structures" on page 2-74 for more information.

Object Method Specifier

Specify the properties of an instance of a MATLAB class using the object name followed by a dot, and then the property name:

```
val = asset.current_value
```

See "MATLAB Classes" on page 2-117 for more information.

Dot-Dot — ..

Two dots in sequence refer to the parent of the current directory.

Parent Directory

Specify the directory immediately above your current directory using two dots. For example, to go up two levels in the directory tree and down into the `testdir` directory, use

```
cd ..\..\testdir
```

Dot-Dot-Dot (Ellipsis) — ...

A series of three consecutive periods (`...`) is the line continuation operator in MATLAB. This is often referred to as an *ellipsis*, but it should be noted that the line continuation operator is a three-character operator and is different from the single-character ellipsis represented in ASCII by the hexadecimal number 2026.

Line Continuation

Continue any MATLAB command or expression by placing an ellipsis at the end of the line to be continued:

```
sprintf('The current value of %s is %d', ...
        vname, value)
```

Entering Long Strings. You cannot use an ellipsis within single quotes to continue a string to the next line:

```
string = 'This is not allowed and will generate an ...
   error in MATLAB.'
```

To enter a string that extends beyond a single line, piece together shorter strings using either the concatenation operator (`[]`) or the `sprintf` function.

Here are two examples:

```
quote1 = [
 'Tiger, tiger, burning bright in the forests of the night,' ...
 'what immortal hand or eye could frame thy fearful symmetry?'];
quote2 = sprintf('%s%s%s', ...
 'In Xanadu did Kubla Khan a stately pleasure-dome decree,', ...
 'where Alph, the sacred river, ran ', ...
 'through caverns measureless to man down to a sunless sea.');
```

Dot-Parentheses — .( )

Use dot-parentheses to specify the name of a dynamic structure field.

Dynamic Structure Fields

Sometimes it is useful to reference structures with field names that can vary. For example, the referenced field might be passed as an argument to a function. Dynamic field names specify a variable name for a structure field.

The variable `fundtype` shown here is a dynamic field name:

```
type = funds(5,2).(fundtype);
```

See “Using Dynamic Field Names” on page 2-80 for more information.

Exclamation Point — !

The exclamation point precedes operating system commands that you want to execute from within MATLAB.

Shell Escape

The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:

```
!rmdir oldtests
```

See “Shell Escape Functions” on page 3-28 for more information.

Parentheses — ( )

Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function.

Array Indexing

When parentheses appear to the right of a variable name, they are indices into the array stored in that variable:

```
A(2, 7, 4)
```

Function Input Arguments

When parentheses follow a function name in a function declaration or call, the enclosed list contains input arguments used by the function:

```
function sendmail(to, subject, message, attachments)
```

Percent — %

The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code. Some functions also interpret the percent sign as a conversion specifier.

See “Help Text” on page 4-11 for more information.

Single Line Comments

Precede any one-line comments in your code with a percent sign. MATLAB does not execute anything that follows a percent sign (that is, unless the sign is quoted, `'%'`):

```
% The purpose of this routine is to compute
% the value of ...
```

Conversion Specifiers

Some functions, like `sscanf` and `sprintf`, precede conversion specifiers with the percent sign:

```
sprintf('%s = %d', name, value)
```

Percent-Brace — %{ %}

The `%{` and `%}` symbols enclose a block of comments that extend beyond one line.

Block Comments

Enclose any multiline comments with percent followed by an opening or closing brace.

```
%{
The purpose of this routine is to compute
the value of ...
%}
```

Note With the exception of whitespace characters, the `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Semicolon — ;

The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.

Array Row Separator

When used within square brackets to create a new array or concatenate existing arrays, the semicolon creates a new row in the array:

```
A = [5, 8; 3, 4]
A =
     5     8
     3     4
```

Output Suppression

When placed at the end of a command, the semicolon tells MATLAB not to display any output from that command. In this example, MATLAB does not display the resulting 100-by-100 matrix:

```
A = ones(100, 100);
```

Command or Statement Separator

Like the comma operator, you can enter more than one MATLAB command on a line by separating each command with a semicolon. MATLAB suppresses output for those commands terminated with a semicolon, and displays the output for commands terminated with a comma.

In this example, assignments to variables `A` and `C` are terminated with a semicolon, and thus do not display. Because the assignment to `B` is comma-terminated, the output of this one command is displayed:

```
A = 12.5;  B = 42.7,  C = 1.25;
B =
   42.7000
```

Single Quotes — ' '

Single quotes are the constructor symbol for MATLAB character arrays.

Character and String Constructor

MATLAB constructs a character array from all characters enclosed in single quotes. If only one character is in quotes, then MATLAB constructs a 1-by-1 array:

```
S = 'Hello World'
```

See "Characters and Strings" on page 2-37 for more information.

Space Character

The space character serves a purpose similar to the comma in that it can be used to separate row elements in an array constructor, or the values returned by a function.

Row Element Separator

You have the option of using either commas or spaces to delimit the row elements of an array when constructing the array. To create a 1-by-3 array, use

```
A = [5.92 8.13 3.53]
A =
    5.9200    8.1300    3.5300
```

When indexing into an array, you must always use commas to reference each dimension of the array.

Function Output Separator

Spaces are allowed when specifying a list of values to be returned by a function. You can use spaces to separate return values in both function declarations and function calls:

```
function [data text] = xlsread(file, sheet, range, mode)
```

Slash and Backslash — / \

The slash (/) and backslash (\) characters separate the elements of a path or directory string. On Windows-based systems, both slash and backslash have the same effect. On UNIX-based systems, you must use slash only.

On a Windows system, you can use either backslash or slash:

```
dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m'])
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

On a UNIX system, use only the forward slash:

```
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

Square Brackets — [ ]

Square brackets are used in array construction and concatenation, and also in declaring and capturing values returned by a function.

Array Constructor

To construct a matrix or array, enclose all elements of the array in square brackets:

```
A = [5.7, 9.8, 7.3; 9.2, 4.5, 6.4]
```

Concatenation

To combine two or more arrays into a new array through concatenation, enclose all array elements in square brackets:

```
A = [B, eye(6), diag([0:2:10])]
```

Function Declarations and Calls

When declaring or calling a function that returns more than one output, enclose each return value that you need in square brackets:

```
[data, text] = xlsread(file, sheet, range, mode)
```

Internal MATLAB Functions

In this section...

"Overview" on page 3-108
"M-File Functions" on page 3-108
"Built-In Functions" on page 3-109
"Overloaded MATLAB Functions" on page 3-110

Overview

Many of the functions provided with MATLAB are implemented as M-files just like the M-files that you will create with MATLAB. Other MATLAB functions are precompiled executable programs called built-ins that run much more efficiently. Many of the MATLAB functions are also overloaded so that they handle different data types appropriately.

M-File Functions

If you look in the subdirectories of the `toolbox\matlab` directory, you can find the M-file sources to many of the functions supplied with MATLAB. You can locate your `toolbox\matlab` directory by typing

```
dir([matlabroot '\toolbox\matlab\'])
```

MATLAB functions with an M-file source are just like any other functions coded with MATLAB. When one of these M-file functions is called, MATLAB parses and executes each line of code in the M-file. It saves the parsed version of the function in memory, eliminating parsing time on any further calls to this function.

Identifying M-File Functions

To find out if a function is implemented with an M-file, use the `exist` function. The `exist` function searches for the name you enter on the MATLAB path and returns a number identifying the source. If the source is an M-file, then `exist` returns the number `2`. This example identifies the source for the `repmat` function as an M-file:

```
exist repmat
ans =
     2
```

The `exist` function also returns 2 for files that have a file type unknown to MATLAB. However, if you invoke `exist` on a MATLAB function name, the file type will be known to MATLAB and will return 2 only on M-files.

Viewing the Source Code

One advantage of functions implemented as M-files is that you can look at the source code. This may help when you need to understand why the function returns a value you did not expect, if you need to figure out how to code something in MATLAB that is already coded in a function, or perhaps to help you create a function that overloads one of the MATLAB functions.

To find the source code for any MATLAB M-file function, use `which`:

```
which repmat
   D:\matlabR14\toolbox\matlab\elmat\repmat.m
```

Built-In Functions

Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called *built-ins*.

Unlike M-file functions, you cannot see the source code for built-ins. Although most built-in functions do have an M-file associated with them, this file is there mainly to supply the help documentation for the function. Take the `reshape` function, for example, and find it on the MATLAB path:

```
which reshape
   D:\matlabR14\toolbox\matlab\elmat\reshape.m
```

If you type this M-file out, you will see that it consists almost entirely of help text. At the bottom is a call to the built-in executable image.

Identifying Built-In Functions

As with M-file functions, you can identify which functions are built-ins using the `exist` function. This function identifies built-ins by returning the number `5`:

```
exist reshape
ans =
     5
```

Forcing a Built-In Call

If you overload any of the MATLAB built-in functions to handle a specific data type, then MATLAB will always call the overloaded function on that type. If, for some reason, you need to call the built-in version, you can override the usual calling mechanism using a function called `builtin`. The expression

```
builtin('reshape', arg1, arg2, ..., argN);
```

forces a call to MATLAB built-in `reshape`, passing the arguments shown even though an overload exists for the data types in this argument list.

Overloaded MATLAB Functions

An overloaded function is an additional implementation of an existing function that has been designed specifically to handle a certain data type. When you pass an argument of this type in a call to the function, MATLAB looks for the function implementation that handles that type and executes that function code.

Each overloaded MATLAB function has an M-file on the MATLAB path. The M-files for a certain data type (or class) are placed in a directory named with an `@` sign followed by the class name. For example, to overload the MATLAB `plot` function to plot expressions of a class named `polynom` differently than other data types, you would create a directory called `@polynom` and store your own version of `plot.m` in that directory.

You can add your own overloads to any function by creating a class directory for the data type you wish to support for that function, and creating an M-file that handles that type in a manner different from the default. See "Setting

Up Class Directories” on page 9-6 and “Designing User Classes in MATLAB” on page 9-9.

When you use the `which` command with the `-all` option, MATLAB returns all occurrences of the file you are looking for. This is an easy way to find functions that are overloaded:

```
which -all set          % Show all implementations for 'set'
```

4

M-File Programming

Program Development (p. 4-2)	Procedures and tools used in creating, debugging, optimizing, and checking in a program
Working with M-Files (p. 4-7)	Introduction to the basic MATLAB program file
M-File Scripts and Functions (p. 4-17)	Overview of scripts, simple programs that require no input or output, and functions, more complex programs that exchange input and output data with the caller
Function Handles (p. 4-22)	Packaging the access to a function into a function handle, and passing that handle to other functions
Function Arguments (p. 4-32)	Handling the data passed into and out of an M-file function, checking input data, passing variable numbers of arguments
Calling Functions (p. 4-52)	Calling syntax, determining which function will be called, passing different types of arguments, passing arguments in structures and cell arrays, identifying function dependencies

Program Development

In this section...

"Overview" on page 4-2

"Creating a Program" on page 4-2

"Getting the Bugs Out" on page 4-3

"Cleaning Up the Program" on page 4-4

"Improving Performance" on page 4-5

"Checking It In" on page 4-6

Overview

When you write a program in MATLAB, you save it to a file called an M-file (named after its `.m` file extension). There are two types of M-files that you can write: scripts and functions. This section covers basic program development, describes how to write and call scripts and functions, and shows how to pass different types of data in a function call. Associated with each step of this process are certain MATLAB tools and utilities that are fully documented in the Desktop Tools and Development Environment documentation.

For more ideas on good programming style, see "Program Development" on page 12-20 in the MATLAB Programming Tips documentation. The Programming Tips section is a compilation of useful pieces of information that can show you alternate and often more efficient ways to accomplish common programming tasks while also expanding your knowledge of MATLAB.

Creating a Program

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose. The Editor/Debugger is fully documented in Ways to Edit and Debug Files in the Desktop Tools and Development Environment documentation.

The first step in creating a program is to open an editing window. To create a new M-file, type the word `edit` at the MATLAB command prompt. To edit an existing M-file, type `edit` followed by the filename:

```
edit drawPlot.m
```

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

Saving the Program

It is usually a good idea to save your program periodically while you are in the development process. To do this, click **File > Save** in the Editor/Debugger. Enter a filename with a `.m` extension in the **Save file as** dialog box that appears and click **OK**. It is customary and less confusing if you give the M-file the same name as the first function in the M-file.

Running the Program

Before trying to run your program, make sure that its M-file is on the MATLAB path. The MATLAB path defines those directories that you want MATLAB to know about when executing M-files. The path includes all the directories that contain functions provided with MATLAB. It should also include any directories that you use for your own functions.

Use the `which` function to see if your program is on the path:

```
which drawPlot
   D:\matlabR14\work\drawPlot.m
```

If not, add its directory to the path using the `addpath` function:

```
addpath('D:\matlabR14\work')
```

Now you can run the program just by typing the name of the M-file at the MATLAB command prompt:

```
drawPlot(xdata, ydata)
```

Getting the Bugs Out

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the command window, programs that hang (never terminate), inaccurate results,

or some number of other symptoms. This is where the second functionality of the MATLAB Editor/Debugger becomes useful.

The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or *breakpoint*) at that line. Once a breakpoint has been set, start your program from the MATLAB command prompt. MATLAB opens an Editor/Debugger window (if it is not already open) showing a green arrow pointing to the next line to execute.

From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

To learn about using the MATLAB Debugger, see Debugging and Improving M-Files in the Desktop Tools and Development Environment documentation. Type `help debug` for a listing of all MATLAB debug functions.

For programming tips on how to debug, see “Debugging” on page 12-23.

Cleaning Up the Program

Even after your program is bug-free, there are still some steps you can take to improve its performance and readability. The MATLAB M-Lint utility generates a report that can highlight potential problems in your code. For example, you may be using the elementwise AND operator (`&`) where the

short-circuit AND (`&&`) is more appropriate. You may be using the `find` function in a context where logical subscripting would be faster.

MATLAB offers M-Lint and several other reporting utilities to help you make the finishing touches to your program code. These tools are described under Tuning and Refining M-Files in the Desktop Tools and Development Environment documentation.

Improving Performance

The MATLAB Profiler generates a report that shows how your program spends its processing time. For details about using the MATLAB Profiler, see Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation. For tips on other ways to improve the performance of your programs, see Chapter 11, “Improving Performance and Memory Usage”.

Three types of reports are available:

- “Summary Report” on page 4-5
- “Detail Report” on page 4-5
- “File Listing” on page 4-6

Summary Report

The summary report provides performance information on your main program and on every function it calls. This includes how many times each function is called, the total time spent in that function, along with a bar graph showing the relative time spent by each function.

Detail Report

When you click a function name in the summary report, MATLAB displays a detailed report on that function. This report shows the lines of that function that take up the most time, the time spent executing that line, the percentage of total time for that function that is spent on that line, and a bar graph showing the relative time spent on the line.

File Listing

The detail report for a function also displays the entire M-file code for that function. This listing enables you to view the time-consuming code in the context of the entire function body. For every line of code that takes any significant time, additional performance information is provided by the statistics and by the color and degree of highlighting of the program code.

Checking It In

Source control systems offer a way to manage large numbers of files while they are under development. They keep track of the work done on these files as your project progresses, and also ensure that changes are made in a secure and orderly fashion.

If you have a source control system available to you, you will probably want to check your M-files into the system once they are complete. If further work is required on one of those files, you just check it back out, make the necessary modifications, and then check it back in again.

MATLAB provides an interface to external source control systems so that you can check files in and out directly from your MATLAB session. See Revision Control in the Desktop Tools and Development Environment documentation for instructions on how to use this interface.

Working with M-Files

In this section...

"Overview" on page 4-7

"Types of M-Files" on page 4-7

"Basic Parts of an M-File" on page 4-8

"Creating a Simple M-File" on page 4-12

"Providing Help for Your Program" on page 4-15

"Creating P-Code Files" on page 4-15

Overview

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of `filename.m`. The term you use for `filename` becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB M-file.

Types of M-Files

M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output.

MATLAB scripts:

- Are useful for automating a series of steps you need to perform many times.
- Do not accept input arguments or return output arguments.
- Store variables in a workspace that is shared with other scripts and with the MATLAB command line interface.

MATLAB functions:

- Are useful for extending the MATLAB language for your application.
- Can accept input arguments and return output arguments.

- Store variables in a workspace internal to the function.

Basic Parts of an M-File

This simple function shows the basic parts of an M-file. Note that any line that begins with % is not executable:

```
function f = fact(n)                        Function
definition line
% Compute a factorial value.                H1 line
% FACT(N) returns the factorial of N,       Help text
% usually denoted by N!

% Put simply, FACT(N) is PROD(1:N).         Comment
f = prod(1:n);                              Function body
```

The table below briefly describes each of these M-file parts. Both functions and scripts can have all of these parts, except for the function definition line which applies to functions only. These parts are described in greater detail following the table.

M-File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request `help` on an entire directory, or when you use `lookfor`
Help text	A more detailed description of the program, displayed together with the H1 line when you request `help` on a specific function
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Function Definition Line

The function definition line informs MATLAB that the M-file contains a function, and specifies the argument calling sequence of the function. The function definition line for the `fact` function is

All MATLAB functions have a function definition line that follows this pattern.

Function Name. Function names must begin with a letter, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed length (returned by the function `namelengthmax`). Because variables must obey similar rules, you can use the `isvarname` function to check whether a function name is valid:

```
isvarname myfun
```

Although function names can be of any length, MATLAB uses only the first `N` characters of the name (where `N` is the number returned by the function `namelengthmax`) and ignores the rest. Hence, it is important to make each function name unique in the first `N` characters:

```
N = namelengthmax
N =
    63
```

Note Some operating systems may restrict file names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension `.m` appended. For example,

```
average.m
```

If the filename and the function definition line name are different, the internal (function) name is ignored. Thus, if `average.m` is the file that defines a function named `computeAverage`, you would invoke the function by typing

```
average
```

Note While the function name specified on the function definition line does not have to be the same as the filename, it is best to use the same name for both to avoid confusion.

Function Arguments. If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses following the function name. Use commas to separate multiple input or output arguments. Here is the declaration for a function named `sphere` that has three inputs and three outputs:

```
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets:

```
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as those in the function definition line.

The H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, `%`. For the `average` function, the H1 line is

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types `help` *`functionname`* at the MATLAB prompt. Further, the `lookfor` function searches on and displays only the H1 line. Because this line provides important summary information about the M-file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your M-files by entering help text on one or more consecutive comment lines at the start of your M-file program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with `%` to be the online help text for the function. The first line without `%` as the left-most character ends the help.

The help text for the `average` function is

```
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
```

When you type `help` *`functionname`* at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Note Help text in an M-file can be viewed at the MATLAB command prompt only (using `help functionname`). You cannot display this text using the MATLAB Help browser. You can, however, use the Help browser to get help on MATLAB functions and also to read the documentation on any MathWorks products.

The Function or Script Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the `average` function contains a number of simple programming statements:

```
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1)) % Flow control
    error('Input must be a vector') % Error message display
end
y = sum(x)/length(x);             % Computation and assignment
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.
y = sum(x)           % Use the sum function.
```

In addition to comment lines, you can insert blank lines anywhere in an M-file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for an M-file.

Block Comments. To write comments that require more than one line, use the block comment operators, `%{` and `%}`:

```
%{
This next block of code checks the number of inputs
passed in, makes sure that each input is a valid data
type, and then branches to start processing the data.
%}
```

Note The `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

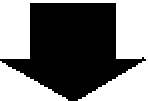
Creating a Simple M-File

You create M-files using a text editor. MATLAB provides a built-in editor, but you can use any text editor you like. Once you have written and saved the M-file, you can run the program as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

```
function c = myfile(a,b)
c = sqrt((a.^2)+(b.^2))
```



2 Call the M-file from the command line, or from within another M-file.

```
a = 7.5
b = 3.342
c = myfile(a,b)

c =

    8.2109
```

Using Text Editors

M-files are ordinary text files that you create using a text editor. If you use the MATLAB Editor/Debugger, open a new file by selecting **New > M-File** from the **File** menu at the top of the MATLAB Command Window.

Another way to edit an M-file is from the MATLAB command line using the `edit` function. For example,

```
edit foo
```

opens the editor on the file `foo.m`. Omitting a filename opens the editor on an untitled file.

You can create the `fact` function shown in “Basic Parts of an M-File” on page 4-8 by opening your text editor, entering the lines shown, and saving the text in a file called `fact.m` in your current directory.

Once you have created this file, here are some things you can:

- List the names of the files in your current directory:

```
what
```

- List the contents of M-file `fact.m`:

```
type fact
```

- Call the `fact` function:

```
fact(5)
ans =
     120
```

A Word of Caution on Saving M-Files

Save any M-files you create and any MathWorks supplied M-files that you edit in directories outside of the directory tree in which the MATLAB software is installed. If you keep your files in any of the installed directories, your files may be overwritten when you install a new version of MATLAB.

MATLAB installs its software into directories under *matlabroot*/`toolbox`. To see what *matlabroot* is on your system, type `matlabroot` at the MATLAB command prompt.

Also note that locations of files in the *matlabroot*/`toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *matlabroot*/`toolbox` directories using an external editor, or if you add or remove files from these directories using file system operations, enter the commands `clear functionname` and `rehash toolbox` before you use the files in the current session.

For more information, see the `rehash` function reference page or the section Toolbox Path Caching in the Desktop Tools and Development Environment documentation.

Providing Help for Your Program

You can provide user information for the programs you write by including a help text section at the beginning of your M-file. (See "Help Text" on page 4-11).

You can also make help entries for an entire directory by creating a file with the special name `Contents.m` that resides in the directory. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
help directoryname
```

`Contents.m` files are optional. You might have directories of your own with M-files that you don't necessarily want public. For this or other reasons, you might choose not to provide this type of help listing for these directories. If you have a directory that is on the path that does not have a `Contents.m` file, MATLAB displays `(No table of contents file)` for that directory in response to the `help` command. If you do not want to see this displayed, creating an empty `Contents.m` file will disable this message for that directory.

Also, if a directory does not contain a `Contents.m` file, typing

```
help directoryname
```

displays the first help line (the H1 line) for each M-file in the directory.

There is a tool in the Current Directory browser, called the Contents Report, that you can use to help create and validate your `Contents.m` files. See Contents File Report in the MATLAB Desktop Tools and Development Environment documentation for more information.

Creating P-Code Files

You can save a preparsed version of a function or script, called P-code files, for later MATLAB sessions using the `pcode` function. For example,

```
pcode average
```

parses `average.m` and saves the resulting pseudocode to the file named `average.p`. This saves MATLAB from reparsing `average.m` the first time you call it in each session.

MATLAB is very fast at parsing so the `pcode` function rarely makes much of a speed difference.

One situation where `pcode` does provide a speed benefit is for large GUI applications. In this case, many M-files must be parsed before the application becomes visible.

You can also use `pcode` to hide algorithms you have created in your M-file, if you need to do this for proprietary reasons.

M-File Scripts and Functions

In this section...

"M-File Scripts" on page 4-17
"M-File Functions" on page 4-18
"Types of Functions" on page 4-19
"Identifying Dependencies" on page 4-20

M-File Scripts

Scripts are the simplest kind of M-file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

The Base Workspace

Scripts share the base workspace with your interactive MATLAB session and with other scripts. They operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations. You should be aware, though, that running a script can unintentionally overwrite data stored in the base workspace by commands entered at the MATLAB command prompt.

Simple Script Example

These statements calculate `rho` for several trigonometric functions of `theta`, then create a series of polar plots:

```
% An M-file script to produce       % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                % Computations
rho(1,:) = 2 * sin(5 * theta) .^ 2;
rho(2,:) = cos(10 * theta) .^ 3;
rho(3,:) = sin(theta) .^ 2;
rho(4,:) = 5 * cos(3.5 * theta) .^ 3;
for k = 1:4
```

```
      polar(theta, rho(k,:))          % Graphics output
      pause
   end
```

Try entering these commands in an M-file called `petals.m`. This file is now a MATLAB script. Typing `petals` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Enter** or **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

M-File Functions

Functions are program routines, usually implemented in M-files, that accept input arguments and return output arguments. They operate on variables within their own workspace. This workspace is separate from the workspace you access at the MATLAB command prompt.

The Function Workspace

Each M-file function has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can, however, define variables as global variables explicitly, allowing more than one workspace context to access them. You can also evaluate any MATLAB statement using variables from either the base workspace or the workspace of the calling function using the `evalin` function. See "Extending Variable Scope" on page 3-10 for more information.

Simple Function Example

The average function is a simple M-file that calculates the average of the elements in a vector:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

Try entering these commands in an M-file called average.m. The average function accepts a single input argument and returns a single output argument. To call the average function, enter

```
z = 1:99;

average(z)
ans =
    50
```

Types of Functions

MATLAB provides the following types of functions. Each function type is described in more detail in a later section of this documentation:

- The “Primary M-File Functions” on page 5-15 is the first function in an M-file and typically contains the main program.
- “Subfunctions” on page 5-33 act as subroutines to the main function. You can also use them to define multiple functions within a single M-file.
- “Nested Functions” on page 5-16 are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the M-file.
- “Anonymous Functions” on page 5-3 provide a quick way of making a function from any MATLAB expression. You can compose anonymous

functions either from within another function or at the MATLAB command prompt.

- "Overloaded Functions" on page 5-37 are useful when you need to create a function that responds to different types of inputs accordingly. They are similar to overloaded functions in any object-oriented language.
- "Private Functions" on page 5-35 give you a way to restrict access to a function. You can call them only from an M-file function in the parent directory.

You might also see the term function functions in the documentation. This is not really a separate function type. The term function functions refers to any functions that accept another function as an input argument. You can pass a function to another function using a function handle.

Identifying Dependencies

Most any program you write will make calls to other functions and scripts. If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

Simple Display of M-File Dependencies

For a simple display of all M-files referenced by a particular function, follow these steps:

1. Type `clear functions` to clear all functions from memory (see Note below).

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions (which you can check using `inmem`) unlock them with `munlock`, and then repeat step 1.

---

2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, because you can get different results when calling the same function with different arguments.

3 Type `inmem` to display all M-files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output:

```
[mfiles, mexfiles] = inmem
```

Detailed Display of M-File Dependencies

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on:

```
[list, builtins, classes] = depfun('strtok.m');

list
list =
   'D:\matlabR14\toolbox\matlab\strfun\strtok.m'
   'D:\matlabR14\toolbox\distcomp\toChar.m'
   'D:\matlabR14\toolbox\matlab\datafun\prod.m'
   'D:\matlabR14\toolbox\matlab\datatypes\@opaque\char.m'
          .
          .
          .
```

Function Handles

In this section...

"Constructing a Function Handle" on page 4-22

"Calling a Function Using Its Handle" on page 4-23

"Functions That Operate on Function Handles" on page 4-25

"Comparing Function Handles" on page 4-25

"Additional Information on Function Handles" on page 4-30

Constructing a Function Handle

A *function handle* is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks).

Use the following syntax to construct a function handle, preceding the name of the function with an @ sign. Use only the function *name*, with no path information, after the @ sign:

```
fhandle = @functionname
```

MATLAB maps the handle to the function you specify and saves this mapping information in the handle. If there is more than one function with this name, MATLAB maps to the one function source it would dispatch to if you were actually calling the function.

A function handle retains that same mapping even if its corresponding function goes out of scope. For example, if, after creating the handle, you change the MATLAB path so that a different function of the same name now takes precedence, invoking the function handle still executes the code to which the handle was originally mapped.

Handles to Anonymous Functions

Another way to construct a function handle is to create an anonymous function. For example,

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument x. The variable sqr contains a handle to the anonymous function. See "Anonymous Functions" on page 5-3 for more information.

Arrays of Function Handles

To store function handles in an array, use a cell array:

```
trigFun = {@sin, @cos, @tan};

plot(trigFun{2}(-pi:0.01:pi))
```

Invalid or Obsolete Function Handles

If you create a handle to a function that is not on the MATLAB path, or if you load a handle to a function that is no longer on the path, MATLAB catches the error only when the handle is invoked. You can assign an invalid handle and use it in such operations as func2str. MATLAB catches and reports an error only when you attempt to use it in a runtime operation.

Calling a Function Using Its Handle

To execute a function associated with a function handle, use the syntax shown here, treating the function handle fhandle as if it were a function name:

```
fhandle(arg1, arg2, ..., argn)
```

If the function being called takes no input arguments, then use empty parentheses after the function handle name:

```
fhandle()
```

Handling Return Values

When you invoke a function by means of its handle, you can capture any or all values returned from the call in the same way you would if you were calling the function directly. Just list the output variable to the left of the

equals sign. When assigning to multiple outputs, enclose the output variables within square brackets:

```
[out1 out2 ...] = fhandle(arg1, arg2, arg3, ...)
```

This example returns multiple values from a call to an anonymous function. Create anonymous function `f` that locates the nonzero elements of an array, and returns the row, column, and value of each element in variables `row`, `col`, and `val`:

```
f = @(X)find(X);
```

Call the function on matrix `m` using the function handle `f`. Because the function uses the MATLAB `find` function which returns up to 3 outputs, you can specify from 0 to 3 outputs in the call:

```
m = [3 2 0; -5 0 7; 0 0 1]
m =
     3     2     0
    -5     0     7
     0     0     1

[row col val] = f(m);

val
val =
     3
    -5
     2
     7
     1
```

Simple Function Handle Example

The following example calls a function `plotFHandle`, passing it a handle for the MATLAB `sin` function. `plotFHandle` then calls the `plot` function, passing it some data and the function handle to `sin`. The plot function calls the function associated with the handle to compute its *y*-axis values:

```
function x = plotFHandle(fhandle, data)
plot(data, fhandle(data))
```

Call `plotFhandle` with a handle to the `sin` function and the value shown below:

```
plotFHandle(@sin, -pi:0.01:pi)
```

Functions That Operate on Function Handles

MATLAB provides the following functions for working with function handles. See the reference pages for these functions for more information.

Function	Description
`functions`	Return information describing a function handle.
`func2str`	Construct a function name string from a function handle.
`str2func`	Construct a function handle from a function name string.
`save`	Save a function handle from the current workspace to a MAT-file.
`load`	Load a function handle from a MAT-file into the current workspace.
`isa`	Determine if a variable contains a function handle.
`isequal`	Determine if two function handles are handles to the same function.

Comparing Function Handles

This section describes how MATLAB determines whether or not separate function handles are equal to each other:

- "Handles Constructed from a Named Function" on page 4-26
- "Handles to Anonymous Functions" on page 4-26
- "Handles to Nested Functions" on page 4-27
- "Handles Saved to a MAT-File" on page 4-28

Handles Constructed from a Named Function

Function handles that you construct from the same named function, e.g., `handle = @sin`, are considered by MATLAB to be equal. The `isequal` function returns a value of `true` when comparing these types of handles:

```
func1 = @sin;
func2 = @sin;
isequal(func1, func2)
 ans =
     1
```

If you save these handles to a MAT-file and then load them back into the workspace later on, they are still equal:

```
save temp1 func1
save temp2 func2
clear

load temp1
load temp2
isequal(func1, func2)
ans =
     1
```

Handles to Anonymous Functions

Unlike handles to named functions, any two function handles that represent the same anonymous function (i.e., handles to anonymous functions that contain the same text) are not equal. This is because MATLAB cannot guarantee that the frozen values of non-argument variables are the same.

```
q = 1;
a1 = @(x)q * x.^2;

q = 2;
a2 = @(x)q * x.^2;

isequal(a1, a2)
ans =
     0
```

This `false` result is accurate because `a1` and `a2` do indeed behave differently.

Note In general, MATLAB may underestimate the equality of function handles. That is, a test for equality may return `false` even when the functions happen to behave the same. But in cases where MATLAB does indicate equality, the functions are guaranteed to behave in an identical manner.

On the other hand, if you make a copy of an anonymous function handle, the copy and the original are equal:

```
h1 = @(x)sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

In this case, function handles `h1` and `h2` are guaranteed to behave identically.

Handles to Nested Functions

Function handles to the same nested function are considered equal only if your code constructs these handles on the same call to the function containing the nested functions. Given this function that constructs two handles to the same nested function,

```
function [h1, h2] = test_eq(a, b, c)
h1 = @findZ;
h2 = @findZ;

   function z = findZ
   z = a.^3 + b.^2 + c';
   end
end
```

any two function handles constructed from the same nested function and on the same call to the parent function are equal:

```
[h1 h2] = test_eq(4, 19, -7);
```

```
isequal(h1, h2)
ans =
     1

[q1 q2] = test_eq(3, -1, 2);
isequal(q1, q2)
ans =
     1
```

The answer makes sense because `h1` and `h2` will always produce the same answer:

```
x = h1(),   y = h2()
x =
   418
y =
   418
```

However, handles constructed on different calls to the parent function are not equal:

```
isequal(h1, q1)
ans =
     0
```

In this case, `h1` and `q1` behave differently:

```
x = h1(),   y = q1()
x =
   418
y =
    30
```

Handles Saved to a MAT-File

If you save equivalent anonymous or nested function handles to separate MAT-files and then load them back into the MATLAB workspace, they are no longer equal. This is because saving the function handle in effect loses track of the original circumstances under which the function handle was created, and reloading it results in a function handle that compares as being unequal to the original function handle.

Create two equivalent anonymous function handles:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

Save each to a different MAT-file:

```
save fname1 h1;
save fname2 h2;
```

Clear the MATLAB workspace and then load the function handles back into the workspace:

```
clear all
load fname1
load fname2
```

The function handles are no longer equal:

```
isequal(h1, h2)
ans =
     0
```

Note however that equal anonymous and nested function handles that you save to the same MAT-file are equal when loaded back into MATLAB:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1

save fname h1 h2;

clear all
load fname
```

```
isequal(h1, h2)
ans =
     1
```

Additional Information on Function Handles

This section covers the following topics:

- "Maximum Length of a Function Name" on page 4-30
- "How MATLAB Constructs a Function Handle" on page 4-30
- "Saving and Loading Function Handles" on page 4-31

Maximum Length of a Function Name

Function names used in handles are unique up to `N` characters, where `N` is the number returned by the function `namelengthmax`. If the function name exceeds that length, MATLAB truncates the latter part of the name.

For function handles created for Java constructors, the length of any segment of the package name or class name must not exceed `namelengthmax` characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, `java.lang.String` has three segments). The overall length of the string specifying the package and class has no limit.

How MATLAB Constructs a Function Handle

At the time you create a function handle, MATLAB maps the handle to one or more implementations of the function specified in the constructor statement:

```
fhandle = @functionname
```

In selecting which function(s) to map to, MATLAB considers

- Scope — The function named must be on the MATLAB path at the time the handle is constructed.
- Precedence — MATLAB selects which function(s) to map to according to the function precedence rules described under "How MATLAB Determines Which Method to Call" on page 9-72.

- Overloading — If additional M-files on the path overload the function for any of the standard MATLAB data types, such as `double` or `char`, then MATLAB maps the handle to these M-files as well.

M-files that overload a function for classes outside of the standard MATLAB data types are not mapped to the function handle at the time it is constructed. Function handles do operate on these types of overloaded functions, but MATLAB determines which implementation to call at the time of evaluation in this case.

Saving and Loading Function Handles

You can save and load function handles in a MAT-file using the MATLAB `save` and `load` functions. If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the M-files that define the function have been moved, and thus no longer exist on the path stored in the handle.
- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either doesn't exist or is located in a different directory than on the system on which the handle was saved.

In both of these cases, the function handle is now invalid because it no longer maps to any existing function code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. Attempting to invoke the handle, however, results in an error.

Function Arguments

In this section...

"Overview" on page 4-32

"Checking the Number of Input Arguments" on page 4-32

"Passing Variable Numbers of Arguments" on page 4-34

"Parsing Inputs with inputParser" on page 4-36

"Passing Optional Arguments to Nested Functions" on page 4-47

"Returning Modified Input Arguments" on page 4-50

Overview

When calling a function, the caller provides the function with any data it needs by passing the data in an argument list. Data that needs to be returned to the caller is passed back in a list of return values.

Semantically speaking, MATLAB always passes argument data by value. (Internally, MATLAB optimizes away any unnecessary copy operations.)

If you pass data to a function that then modifies this data, you will need to update your own copy of the data. You can do this by having the function return the updated value as an output argument.

Checking the Number of Input Arguments

The `nargin` and `nargout` functions enable you to determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a, b)
if (nargin == 1)
    c = a .^ 2;
elseif (nargin == 2)
    c = a + b;
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here is a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by white space or some other character. Given one input, the function assumes a default delimiter of white space; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists:

```
function [token, remainder] = strtok(string, delimiters)
% Function requires at least one input argument
if nargin < 1
   error('Not enough input arguments.');
end
token = []; remainder = [];
len = length(string);
if len == 0
   return
end

% If one input, use white space delimiter
if (nargin == 1)
   delimiters = [9:13 32]; % White space characters
end
i = 1;

% Determine where nondelimiter characters begin
while (any(string(i) == delimiters))
   i = i + 1;
   if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))
   i = i + 1;
   if (i > len), break, end
end
finish = i - 1;
token = string(start:finish);
```

```
% For two output arguments, count characters after
% first delimiter (remainder)
if (nargout == 2)
   remainder = string(finish+1:end);
end
```

The `strtok` function is a MATLAB M-file in the `strfun` directory.

Note The order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Variable Numbers of Arguments

The `varargin` and `varargout` functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers

- "Unpacking varargin Contents" on page 4-35
- "Packing varargout Contents" on page 4-35
- "varargin and varargout in Argument Lists" on page 4-36

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data — one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

Here is an example function that accepts any number of two-element vectors and draws a line to connect them:

```
function testvar(varargin)
for k = 1:length(varargin)
   x(k) = varargin{k}(1); % Cell array indexing
   y(k) = varargin{k}(2);
```

```
end
xmin = min(0,min(x));
ymin = min(0,min(y));
axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
plot(x,y)
```

Coded this way, the `testvar` function works with various input lists; for example,

```
testvar([2 3],[1 5],[4 8],[6 5],[4 2],[2 3])
testvar([-1 0],[3 -5],[4 2],[1 1])
```

Unpacking varargin Contents

Because `varargin` contains all the input arguments in a cell array, it's necessary to use cell array indexing to extract the data. For example,

```
y(n) = varargin{n}(2);
```

Cell array indexing has two subscript components:

- The indices within curly braces `{}` specify which cell to get the contents of.
- The indices within parentheses `()` specify a particular element of that cell.

In the preceding code, the indexing expression `{i}` accesses the `n`th cell of `varargin`. The expression `(2)` represents the second element of the cell contents.

Packing varargout Contents

When allowing a variable number of output arguments, you must pack all of the output into the `varargout` cell array. Use `nargout` to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of `x` coordinates and the second represents `y` coordinates. It breaks the array into separate `[xi yi]` vectors that you can pass into the `testvar` function shown at the beginning of the section on "Passing Variable Numbers of Arguments" on page 4-34:

```
function [varargout] = testvar2(arrayin)
```

```
for k = 1:nargout
   varargout{k} = arrayin(k,:);   % Cell array assignment
end
```

The assignment statement inside the `for` loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see "Cell Arrays" on page 2-93.

To call `testvar2`, type

```
a = [1 2; 3 4; 5 6; 7 8; 9 0];

[p1, p2, p3, p4, p5] = testvar2(a)
p1 =
     1     2
p2 =
     3     4
p3 =
     5     6
p4 =
     7     8
p5 =
     9     0
```

varargin and varargout in Argument Lists

`varargin` or `varargout` must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of `varargin` and `varargout`:

```
function [out1,out2] = example1(a,b,varargin)
function [i,j,varargout] = example2(x1,y1,x2,y2,flag)
```

Parsing Inputs with inputParser

MATLAB provides a class called `inputParser` to handle the different types of arguments passed into an M-file function. Using `inputParser`, you create a schema that both represents and verifies the content of the entire list of

input arguments passed on a call to the function. When used in all of your code development, this schema offers a consistent and thorough means of managing and validating the input information.

This section covers the following topics

- "Defining a Specification for Each Input Parameter" on page 4-37
- "Parsing Parameter Values on the Function Call" on page 4-40
- "Packaging Arguments in a Structure" on page 4-41
- "Arguments That Default" on page 4-43
- "Validating the Input Arguments" on page 4-43
- "Making a Copy of the Schema" on page 4-46
- "Summary of inputParser Methods" on page 4-46
- "Summary of inputParser Properties that Control Parsing" on page 4-46
- "Summary of inputParser Properties that Provide Information" on page 4-47

To illustrate how to use the inputParser class, the documentation in this section develops a new M-file program called `publish_ip`, (based on the MATLAB `publish` function). There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

There is one required argument (`script`), one optional argument (`format`), and a number of optional arguments that are specified as parameter-value pairs (`options`).

Defining a Specification for Each Input Parameter

Most programs have a block of code toward the beginning that parses the values in the input argument list and checks these values against what is expected. The `inputParser` class provides the following methods with which you can specify what the inputs are and whether they are required, optional, or to be specified using the parameter-value syntax:

- addRequired — Add a required parameter to the schema
- addOptional — Add an optional parameter to the schema
- addParamValue — Add an optional parameter-value pair to the schema

Creating the inputParser Object. Call the class constructor for inputParser to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class.

Begin writing the example publish_ip M-file by entering the following two statements:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.
```

After calling the constructor, use the addRequired, addOptional, and addParamValue methods to add arguments to the schema.

Note The constructor and all methods and properties of the inputParser class are case sensitive.

Adding Arguments to the Schema. Add any required arguments to the schema using the addRequired method. This method takes two inputs: the name of the required parameter, and an optional handle to a function that validates the parameter:

```
addRequired(name, validator);
```

Put an addRequired statement at the end of your publish_ip code. The two arguments for addRequired in this case are the filename input, script, and a handle to a function that will validate the filename, ischar. After adding the addRequired statement, your publish_ip function should now look like this:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);
```

Use the `addOptional` method to add any arguments that are not required. The syntax for `addOptional` is similar to that of `addRequired` except that you also need to specify a default value to be used whenever the optional argument is not passed:

```
addOptional(name, default, validator);
```

In this case, the `validator` input is a handle to an anonymous function:

```
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
```

Use `addParamValue` to specify any arguments that use a parameter-value format. The syntax is

```
addParamValue(name, default, validator);
```

For example,

```
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Listing the Arguments. At this point, the schema is complete. Here is the file `publish_ip.m`:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);

p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));p.

p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

When you call the program, MATLAB stores the name of each argument in the `Parameters` property of object `p`. Add the following two lines to your `publish_ip` M-file to display `p.Parameters`:

```
sprintf('%s\n  %s\n  %s\n  %s\n  %s\n  %s', ...
   'The input parameters for this program are:', ...
   p.Parameters{:})
```

Save the M-file, and then run it as shown here:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);
```

The output is

```
The input parameters for this program are:
  format
  maxHeight
  maxWidth
  outputDir
  script
```

Parsing Parameter Values on the Function Call

Once you have constructed a schema that represents all possible inputs to the function, the next task is to write the code that parses and verifies these arguments whenever the function is called. The `parse` method of the `inputParser` class reads and validates the required `script` argument and any optional arguments that follow it in the argument list:

```
p.parse(script, varargin{:});
```

Execution of the parse method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of all arguments, type

```
p.Results
```

To get the value of any single input argument, type

```
p.Results.argname
```

where `argname` is the name of the argument. Continue with the `publish_ip` exercise started earlier in this section by removing the `sprintf` statement that was inserted in the last section, and then adding the following lines:

```
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value of a specific argument.
disp(' ')
disp(sprintf('\nThe maximum height is %d.', ...
     p.Results.maxHeight))

% Display all arguments.
disp(' ')
disp 'List of all arguments:'
disp(p.Results)
```

Now save and execute the M-file, passing the required script argument, the optional format argument, as well as several parameter-value arguments. MATLAB assigns those values you pass in the argument list to the appropriate fields of the `Results` structure:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
              format: 'ppt'
           maxHeight: 300
            maxWidth: 500
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

Packaging Arguments in a Structure

By setting the `StructExpand` property of the `inputParser` object to `true`, you can pass arguments to your function in the form of a structure instead of individually in the argument list. This property must be set prior to calling the `parse` method.

`StructExpand` defaults to the `true` state, so you don't have to make any changes to your test program to run this example.

Put all of the input arguments into a structure:

```
s.format = 'xml';
s.outputDir = 'C:/matlab/test';
s.maxWidth = 200;
s.maxHeight = 150;
```

Now call the function, passing the filename and input structure:

```
publish_ip('ipscript.m', s);

The maximum height is 150.

List of all arguments:
              format: 'xml'
           maxHeight: 150
            maxWidth: 200
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

To disable struct expansion, include the following statement somewhere in your program code before the `p.parse` statement:

```
p.StructExpand = false;
```

Overriding the Input Structure. If you want to pass your argument list in a structure, as described in the previous section, but you also want to alter the value of one or more of these arguments without having to modify the structure, you can do so by passing both the structure and the modified argument:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp');

List of all arguments:
              format: 'xml'
           maxHeight: 150
            maxWidth: 200
           outputDir: 'C:/matlab/R2007a/temp'
              script: 'ipscript.m'
```

Arguments That Default

Any arguments that you do not include in a call to your function are given their default values by MATLAB. You defined these default values when you created your schema using the `addOptional` and `addParamValue` methods. The `UsingDefaults` property is actually a structure that contains the names of any arguments that were not passed in the function call, and thus were assigned default values.

Add the following to your M-file:

```
% Show which arguments were not specified in the call.
disp(' ')
disp 'List of arguments given default values:'
for k=1:numel(p.UsingDefaults)
   field = char(p.UsingDefaults(k));
   value = p.Results.(field);
   if isempty(value),   value = '[]';   end
   disp(sprintf('   ''%s''    defaults to %s', field, value))
end
```

Save the M-file and run it without specifying the `format`, `outputDir`, or `maxHeight` arguments:

```
publish_ip('ipscript.m', 'maxWidth', 500);

List of arguments given default values:
   'format'    defaults to html
   'outputDir'    defaults to D:\work_r14
   'maxHeight'    defaults to []
```

Validating the Input Arguments

When you call your function, MATLAB checks any arguments for which you have specified a validator function. If the validator finds an error, MATLAB displays an error message and aborts the function. In the `publish` function example, the `outputDir` argument validates the value passed in using `@ischar`.

Pass a number instead of a string for the `outputDir` argument:

```
publish_ip('ipscript.m', 'outputDir', 5);
??? Argument 'outputDir' failed validation ischar.

Error in ==> publish_ip at 14
p.parse(varargin{:});
```

Handling Unmatched Arguments. MATLAB throws an error if you call your function with any arguments that are not part of the `inputParser` schema. You can disable this error by setting the `KeepUnmatched` property to `true`. When `KeepUnmatched` is in the `true` state, MATLAB does not throw an error, but instead stores any arguments that are not in the schema in a cell array of strings accessible through the `Unmatched` property of the object. `KeepUnmatched` defaults to `false`.

At some point in your `publish_ip` M-file before executing the `parse` method, set the `KeepUnmatched` property to `true`, and following the `parse` statement, examine the `Unmatched` property:

```
p.KeepUnmatched = true;

% Parse and validate all input arguments.
p.parse(script, varargin{:});

disp(' ')
disp 'List of unmatched arguments:'
p.Unmatched
```

Save and run the function, passing two arguments that are not defined in the schema:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp', ...
           'colorSpace', 'CMYK', 'density', 200);

List of unmatched arguments:
    colorSpace: 'CMYK'
       density: 200
```

Enabling Case-Sensitive Matching. When you pass optional arguments in the function call, MATLAB compares these arguments with the names of parameter-value argument names in the schema. By default, MATLAB does not use case sensitivity in this comparison. So, an argument name entered into the schema (using `addParamValue`) as `maxHeight` will match an argument passed as `MAXHeight` in the function call. You can override the default and make these comparisons case sensitive by setting the `CaseSensitive` property of the object to `true`. MATLAB does not error on a case mismatch, however, unless the `KeepUnmatched` property is set to `false`: its default state.

At some point in your `publish_ip` M-file before executing the `parse` method, set `KeepUnmatched` to `false` and `CaseSensitive` to `true`, and then execute the `publish_ip` function using `MAXHeight` as the name of the argument for specifying maximum height:

```
p.KeepUnmatched = false;
p.CaseSensitive = true;

% Parse and validate all input arguments.
p.parse(script, varargin{:});
```

Save and run the function, using `MAXHeight` as the name of the argument for specifying maximum height:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'MAXHeight', 300);
??? Argument 'MAXHeight' did not match any valid parameter of
the parser.

Error in ==> publish_ip at 17
```

Adding the Function Name to Error Messages. Use the `FunctionName` property to include the name of your function in error messages thrown by the validating function:

At some point in your `publish_ip` M-file before executing the `parse` method, set the `FunctionName` property to `PUBLISH_IP`, and then run the function:

```
p.FunctionName = 'PUBLISH_IP';

% Parse and validate all input arguments.
```

```
p.parse(script, varargin{:});
```

Save and run the function and observe text of the error message:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 5, ...
    'maxWidth', 500, 'maxHeight', 300);
??? Argument 'outputDir' failed validation ischar in PUBLISH_IP.
```

Making a Copy of the Schema

The `createCopy` method enables you to make a copy of an existing schema. Because the `inputParser` class uses handle semantics, you cannot make a copy of the object using an assignment statement.

The following statement creates an `inputParser` object `s` that is a copy of `p`:

```
s = p.createCopy
```

Summary of inputParser Methods

Method	Description
`addOptional`	Add an optional argument to the schema
`addParamValue`	Add a parameter-value pair argument to the schema
`addRequired`	Add a required argument to the schema
`createCopy`	Create a copy of the `inputParser` object
`parse`	Parse and validate the named inputs

Summary of inputParser Properties that Control Parsing

Property	Description
`CaseSensitivity`	Enable or disable case-sensitive matching of argument names. Defaults to `false`.

Property	Description
FunctionName	Function name to be included in error messages. Defaults to an empty string.
KeepUnmatched	Enable or disable errors on unmatched arguments. Defaults to false.
StructExpand	Enable or disable passing arguments in a structure. Defaults to true.

Summary of inputParser Properties that Provide Information

Property	Description
Parameters	Names of arguments defined in inputParser schema.
Results	Names and values of arguments passed in function call that are in the schema for this function.
Unmatched	Names and values of arguments passed in function call that are not in the schema for this function.
UsingDefaults	Names of arguments not passed in function call that are given default values.

Passing Optional Arguments to Nested Functions

You can use optional input and output arguments with nested functions, but you should be aware of how MATLAB interprets varargin, varargout, nargin, and nargout under those circumstances.

varargin and varargout are variables and, as such, they follow exactly the same scoping rules as any other MATLAB variable. Because nested functions share the workspaces of all outer functions, varargin and varargout used in a nested function can refer to optional arguments passed to or from the nested function, or passed to or from one of its outer functions.

nargin and nargout, on the other hand, are functions and when called within a nested function, always return the number of arguments passed to or from the nested function itself.

Using varargin and varargout

varargin or varargout used in a nested function can refer to optional arguments passed to or from that function, or to optional arguments passed to or from an outer function.

- If a nested function includes varargin or varargout in its function declaration line, then the use of varargin or varargout within that function returns optional arguments passed to or from that function.
- If varargin or varargout are not in the nested function declaration but are in the declaration of an outer function, then the use of varargin or varargout within the nested function returns optional arguments passed to the outer function.

In the example below, function C is nested within function B, and function B is nested within function A. The term varargin{1} in function B refers to the second input passed to the primary function A, while varargin{1} in function C refers to the first argument, z, passed from function B:

```
function x = A(y, varargin)   % Primary function A
B(nargin, y * rand(4))

   function B(argsIn, z)      % Nested function B
   if argsIn >= 2
      C(z, varargin{1}, 4.512, 1.729)
   end

      function C(varargin)    % Nested function C
      if nargin >= 2
         x = varargin{1}
      end
      end   % End nested function C
   end   % End nested function B
end   % End primary function A
```

Using nargin and nargout

When nargin or nargout appears in a nested function, it refers to the number of inputs or outputs passed to that particular function, regardless of whether or not it is nested.

In the example shown above, nargin in function A is the number of inputs passed to A, and nargin in function C is the number of inputs passed to C. If a nested function needs the value of nargin or nargout from an outer function, you can pass this value in as a separate argument, as done in function B.

Example of Passing Optional Arguments to Nested Functions

This example references the primary function's varargin cell array from each of two nested functions. (Because the workspace of an outer function is shared with all functions nested within it, there is no need to pass varargin to the nested functions.)

Both nested functions make use of the nargin value that applies to the primary function. Calling nargin from the nested function would return the number of inputs passed to that nested function, and not those that had been passed to the primary. For this reason, the primary function must pass its nargin value to the nested functions.

```
function meters = convert2meters(miles, varargin)
% Converts MILES (plus optional FEET and INCHES input)
% values to METERS.

if nargin < 1 || nargin > 3
   error('1 to 3 input arguments are required');
end

   function feet = convert2Feet(argsIn)
   % Nested function that converts miles to feet and adds in
   % optional FEET argument.

   feet = miles .* 5280;

   if argsIn >= 2
      feet = feet + varargin{1};
   end
```

```
   end  % End nested function convert2Feet

   function inches = convert2Inches(argsIn)
   % Nested function that converts feet to inches and adds in
   % optional INCHES argument.

   inches = feet .* 12;

   if argsIn == 3
      inches = inches + varargin{2};
   end
   end  % End nested function convert2Inches

feet = convert2Feet(nargin);
inches = convert2Inches(nargin);

meters = inches .* 2.54 ./ 100;
end  % End primary function convert2meters

convert2meters(5)
ans =
  8.0467e+003

convert2meters(5, 2000, 4.7)
ans =
  8.6564e+003
```

Returning Modified Input Arguments

If you pass any input variables that the function can modify, you will need to include the same variables as output arguments so that the caller receives the updated value.

For example, if the function `readText`, shown below, reads one line of a file each time is it called, then it must keep track of the offset into the file. But when `readText` terminates, its copy of the `offset` variable is cleared from memory. To keep the offset value from being lost, `readText` must return this value to the caller:

```
function [text, offset] = readText(filestart, offset)
```

Calling Functions

In this section...

"What Happens When You Call a Function" on page 4-52
"Determining Which Function Is Called" on page 4-53
"MATLAB Calling Syntax" on page 4-56
"Passing Certain Argument Types" on page 4-60
"Passing Arguments in Structures or Cell Arrays" on page 4-62
"Assigning Output Arguments" on page 4-64
"Calling External Functions" on page 4-66
"Running External Programs" on page 4-67

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using the `clear` function, or until you quit MATLAB.

Clearing Functions from Memory

You can use `clear` in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
`clear functionname`	Remove specified function from workspace.
`clear functions`	Remove all compiled M-functions.
`clear all`	Remove all variables and functions.

Determining Which Function Is Called

When more than one function has the same name, which one does MATLAB call? This section explains the process that MATLAB uses to make this decision. It covers the following topics:

- "Function Scope" on page 4-53
- "Precedence Order" on page 4-53
- "Multiple Implementation Types" on page 4-55
- "Querying Which Function MATLAB Will Call" on page 4-55

Also keep in mind that there are certain situations in which function names can conflict with *variables* of the same name. See "Potential Conflict with Function Names" on page 3-7for more information.

Function Scope

Any functions you call must first be within the scope of (i.e., visible to) the calling function or your MATLAB session. MATLAB determines if a function is in scope by searching for the function's executable file according to a certain order (see "Precedence Order" on page 4-53).

One key part of this search order is the MATLAB path. The path is an ordered list of directories that MATLAB defines on startup. You can add or remove any directories you want from the path. MATLAB searches the path for the given function name, starting at the first directory in the path string and continuing until either the function file is found or the list of directories is exhausted. If no function of that name is found, then the function is considered to be out of scope and MATLAB issues an error.

Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules in the order given here.

For items 3 through 7 in this list, the file MATLAB searches for can be any of four types: an M- or built-in file, preparsed M-file (P-Code), compiled

C or Fortran file (MEX-file), or Simulink® model (MDL-file). See "Multiple Implementation Types" on page 4-55 for more on this.

1. Variable

   Before assuming that a name should match a function, MATLAB checks the current workspace to see if it matches a variable name. If MATLAB finds a match, it stops the search.

2. Subfunction

   Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

3. Private function

   Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

4. Class constructor

   Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

5. Overloaded method

   MATLAB calls an overloaded method if it is not superseded by a subfunction or private function. Which overloaded method gets called depends on the classes of the objects passed in the argument list.

6. Function in the current directory

   A function in the current working directory is selected before one elsewhere on the path.

7 Function elsewhere on the path

Finally, a function elsewhere on the path is selected. A function in a directory that is toward the beginning of the path string is given higher precedence.

Note Because variables have the highest precedence, if you have created a variable of the same name as a function, MATLAB will not be able to run that function until you clear the variable from memory.

Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is

1. Built-in file
2. MEX-files
3. MDL (Simulink® model) file
4. P-code file
5. M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Function MATLAB Will Call

You can determine which function MATLAB will call using the `which` command. For example,

```
which pie3
matlabroot/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m      % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

MATLAB Calling Syntax

This section explains how to use the MATLAB command and function syntax:

- "MATLAB Command Syntax" on page 4-56
- "MATLAB Function Syntax" on page 4-57
- "Passing Arguments with Command and Function Syntax" on page 4-57
- "How MATLAB Recognizes Function Calls That Use Command Syntax" on page 4-59

You can call function M-files from either the MATLAB command line or from within other M-files. Be sure to include all necessary arguments, enclosing input arguments in parentheses and output arguments in square brackets.

Note Function names are sensitive to case. When you call a function, use the correct combination of upper and lowercase letters so that the name is an exact match. Otherwise, you risk calling a different function that does match but is elsewhere on the path.

You often have the choice of using one of two syntaxes for a function call. You can use either a command or a function type of syntax. This is referred to in MATLAB as *command/function duality*.

MATLAB Command Syntax

A function call made in command syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname in1 in2 ... inN
```

While the command syntax is simpler to write, it has the restriction that you may not assign any return values the function might generate. Attempting to do so generates an error.

Two examples of command syntax are

```
save mydata.mat x y z
clear length width depth
```

In the command syntax, MATLAB treats all arguments as string literals.

MATLAB Function Syntax

Function calls written in the function syntax look essentially the same as those in many other programming languages. One difference is that, in MATLAB, functions can return more than one output value.

A function call with a single return value looks like this:

```
out = functionname(in1, in2, ..., inN)
```

If the function returns more than one value, separate the output variables with commas or spaces, and enclose them all in square brackets (`[]`):

```
[out1, out2, ..., outN] = functionname(in1, in2, ..., inN)
```

Here are two examples:

```
copyfile(srcfile, '..\mytests', 'writable')
[x1, x2, x3, x4] = deal(A{:})
```

In the function syntax, MATLAB passes arguments to the function by value. See the examples under "Passing Arguments with Command and Function Syntax" on page 4-57.

Passing Arguments with Command and Function Syntax

When you call a function using function syntax, MATLAB passes the *values* assigned to each variable in the argument list. For example, this expression passes the values assigned to `A0`, `A1`, and `A2` to the `polyeig` function:

```
e = polyeig(A0, A1, A2)
```

Function calls written in command syntax pass all arguments as string literals. This expression passes the strings 'mydata.mat', 'x', 'y', and 'z' to the save function:

```
save mydata.mat x y z
```

The following examples show the difference between passing arguments in the two syntaxes.

Passing Arguments — Example 1. Calling disp with the function syntax, disp(A), passes the value of variable A to the disp function:

```
A = pi;

disp(A)                       % Function syntax
   3.1416
```

Calling it with the command syntax, disp A, passes the string 'A':

```
A = pi;

disp A                        % Command syntax
   A
```

Passing Arguments — Example 2. Passing two variables representing equal strings to the strcmp function using function and command syntaxes gives different results. The function syntax passes the values of the arguments. strcmp returns a 1, which means they are equal:

```
str1 = 'one';       str2 = 'one';

strcmp(str1, str2)                     % Function syntax
ans =
   1       (equal)
```

The command syntax passes the names of the variables, 'str1' and 'str2', which are unequal:

```
str1 = 'one';       str2 = 'one';
```

```
strcmp str1 str2                    % Command syntax
ans =
   0        (unequal)
```

How MATLAB Recognizes Function Calls That Use Command Syntax

It can be difficult to tell whether a MATLAB expression is a function call using command syntax or another kind of expression, such as an operation on one or more variables. Consider the following example:

```
ls ./d
```

Is this a call to the `ls` function with the directory `./d` as its argument? Or is it a request to perform elementwise division on the array that is the value of the `ls` variable, using the value of the `d` variable as the divisor?

This example might appear unambiguous because MATLAB can determine whether `ls` and `d` are functions or variables. But that is not always true. Some MATLAB components, such as M-Lint and the Editor/Debugger, must operate without reference to the MATLAB path or workspace. MATLAB therefore uses syntactic rules to determine when an expression is a function call using command syntax.

The rules are complicated and have exceptions. In general, when MATLAB recognizes an identifier (which might name a function or a variable), it analyzes the characters that follow the identifier to determine what kind of expression exists. The expression is usually a function call using command syntax when all of the following are true:

1. The identifier is followed immediately by white space.
2. The characters following the white space are not parentheses or an assignment operator.
3. The characters following the white space are not an operator that is itself followed by additional white space and then by characters that can legitimately follow an operator.

The example above meets all three criteria and is therefore a function call using command syntax:

```
ls ./d
```

The following examples are not function calls using command syntax:

```
% No white space following the ls identifier
% Interpretation: elementwise division
ls./d

% Parenthesis following white space
% Interpretation: function call using function syntax
ls ('./d')

% Assignment operator following white space
% Interpretation: assignment to a variable
ls =d

% Operator following white space, followed in turn by
% more white space and a variable
% Interpretation: elementwise division
ls ./ d
```

Passing Certain Argument Types

This section explains how to pass the following types of data in a function call:

- “Passing Strings” on page 4-60
- “Passing Filenames” on page 4-61
- “Passing Function Handles” on page 4-62

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, (`'string'`). For example, to create a new directory called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
dirname = 'myapptests';
mkdir(dirname)
```

Passing Filenames

You can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`):

```
load mydata.mat          % Command syntax
load('mydata.mat')       % Function syntax
```

If you assign the output to a variable, you must use the function syntax:

```
savedData = load('mydata.mat')
```

Specify ASCII files as shown here. In this case, the file extension is required:

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time. There are several ways that your function code can work on specific files without your having to hardcode their filenames into the program. You can

- Pass the filename as an argument:

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function:

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function:

  ```
  [filename, pathname] = uigetfile('*.mat', 'Select MAT-file');
  ```

Passing Function Handles

The MATLAB function handle has several uses, the most common being a means of immediate access to the function it represents. You can pass function handles in argument lists to other functions, enabling the receiving function to make calls by means of the handle.

To pass a function handle, include its variable name in the argument list of the call:

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The receiving function invokes the function being passed using the usual MATLAB calling syntax:

```
function [xf, fval, exitflag, output] = ...
         fminbnd(fhandle, ax, bx, options, varargin)
            .
            .
            .
113  fx = fhandle(x, varargin{:});
```

Passing Arguments in Structures or Cell Arrays

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure or cell array.

Passing Arguments in a Structure

Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields. Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

This example updates weather statistics from information in the following chart.

City	Temp.	Heat Index	Wind Speed	Wind Chill
Boston	43	32	8	37
Chicago	34	27	3	30
Lincoln	25	17	11	16
Denver	15	-5	9	0
Las Vegas	31	22	4	35
San Francisco	52	47	18	42

The information is stored in structure W. The structure has one field for each column of data:

```
W = struct('city', {'Bos','Chi','Lin','Dnv','Vgs','SFr'}, ...
           'temp', {43, 34, 25, 15, 31, 52}, ...
           'heatix', {32, 27, 17, -5, 22, 47}, ...
           'wspeed', {8, 3, 11, 9, 4, 18}, ...
           'wchill', {37, 30, 16, 0, 35, 42});
```

To update the data base, you can pass the entire structure, or just one field with its associated data. In the call shown here, W.wchill is a comma-separated list:

```
updateStats(W.wchill);
```

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The advantage over structures is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function. The disadvantage is that you don't have field names to describe each variable.

This example passes several attribute-value arguments to the plot function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';         C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';   C{2,2} = 'k';
```

```
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Assigning Output Arguments

Use the syntax shown here to store any values that are returned by the function you are calling. To store one output, put the variable that is to hold that output to the left of the equal sign:

```
vout = myfun(vin1, vin2, ...);
```

To store more than one output, list the output variables inside square brackets and separate them with commas or spaces:

```
[vout1 vout2 ...] = myfun(vin1, vin2, ...);
```

The number of output variables in your function call statement does not have to match the number of return values declared in the function being called. For a function that declares N return values, you can specify anywhere from zero to N output variables in the call statement. Any return values that you do not have an output variable for are discarded.

Functions return output values in the order in which the corresponding output variables appear in the function definition line within the M-file. This function returns 100 first, then `x * y`, and lastly `x.^2`:

```
function [a b c] = myfun(x, y)
b = x * y;    a = 100;    c = x.^2;
```

If called with only one output variable in the call statement, the function returns only 100 and discards the values of `b` and `c`. If called with no outputs, the functions returns 100 in the MATLAB default variable `ans`.

Assigning Optional Return Values

The section "Passing Variable Numbers of Arguments" on page 4-34 describes the method of returning optional outputs in a cell array called `varargout`. A function that uses `varargout` to return optional values has a function definition line that looks like one of the following:

```
function varargout = myfun(vin1, vin2, ...)
```

```
function [vout1 vout2 ... varargout] = myfun(vin1, vin2, ...)
```

The code within the function builds the varargout cell array. The content and order of elements in the cell array determines how MATLAB assigns optional return values to output variables in the function call.

In the case where varargout is the only variable shown to the left of the equal sign in the function definition line, MATLAB assigns varargout{1} to the first output variable, varargout{2} to the second, and so on. If there are other outputs declared in the function definition line, then MATLAB assigns those outputs to the leftmost output variables in the call statement, and then assigns outputs taken from the varargout array to the remaining output variables in the order just described.

This function builds the varargout array using descending rows of a 5-by-5 matrix. The function is capable of returning up to six outputs:

```
function varargout = byRow(a)
varargout{1} = '    With VARARGOUT constructed by row ...';
for k = 1:5
    row = 5 - (k-1);              % Reverse row order
    varargout{k+1} = a(row,:);
end
```

Call the function, assigning outputs to four variables. MATLAB returns varargout{1:4}, with rows of the matrix in varargout{2:4} and in the order in which they were stored by the function:

```
[text r1 r2 r3] = byRow(magic(5))
text =
    With VARARGOUT constructed by row ...
r1 =
    11    18    25     2     9
r2 =
    10    12    19    21     3
r3 =
     4     6    13    20    22
```

A similar function builds the varargout array using diagonals of a 5-by-5 matrix:

```
function varargout = byDiag(a)
varargout{1} = '    With VARARGOUT constructed by diagonal ...';
for k = -4:4
    varargout{k + 6} = diag(a, k);
end
```

Call the function with five output variables. Again, MATLAB assigns elements of `varargout` according to the manner in which it was constructed within the function:

```
[text d1 d2 d3 d4] = byDiag(magic(5))
text =
    With VARARGOUT constructed by diagonal ...
d1 =
    11
d2 =
    10
    18
d3 =
     4
    12
    25
d4 =
    23
     6
    19
     2
```

Calling External Functions

The MATLAB external interface offers a number of ways to run external functions from MATLAB. This includes programs written in C or Fortran, methods invoked on Java or COM (Component Object Model) objects, functions that interface with serial port hardware, and functions stored in shared libraries. The *MATLAB External Interfaces* documentation describes these various interfaces and how to call these external functions.

Running External Programs

For information on how to invoke operating systems commands or execute programs that are external to MATLAB, see Running External Programs in the MATLAB Desktop Tools and Development documentation.

5

Types of Functions

Overview of MATLAB Function Types (p. 5-2)	An introduction to the basic types of functions available with MATLAB
Anonymous Functions (p. 5-3)	Functions defined from a MATLAB expression and without requiring an M-file
Primary M-File Functions (p. 5-15)	The first, and often the main, function in an M-file
Nested Functions (p. 5-16)	Functions defined within the body of another function
Subfunctions (p. 5-33)	Any functions that follow the primary function in an M-file
Private Functions (p. 5-35)	Functions with restricted access, callable only from an M-file function in the parent directory
Overloaded Functions (p. 5-37)	Functions with multiple implementations that respond to different types of inputs accordingly

Overview of MATLAB Function Types

There are essentially two ways to create a new function in MATLAB: in a command entered at run-time, or in a file saved to permanent storage.

The command-oriented function, called an *anonymous function*, is relatively brief in its content. It consists of a single MATLAB statement that can interact with multiple input and output arguments. The benefit of using anonymous functions is that you do not have to edit and maintain a file for functions that require only a brief definition.

There are several types of functions that are stored in files (called M-files). The most basic of these are *primary functions* and *subfunctions*. Primary functions are visible to other functions outside of their M-file, while subfunctions, generally speaking, are not. That is, you can call a primary function from an anonymous function or from a function defined in a separate M-file, whereas you can call a subfunction only from functions within the same M-file. (See the Description section of the `function_handle` reference page for information on making a subfunction externally visible.)

Two specific types of primary M-file functions are the *private* and *overloaded function*. Private functions are visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function. Overloaded functions act the same way as overloaded functions in most computer languages. You can create multiple implementations of a function so that each responds accordingly to different types of inputs.

The last type of MATLAB function is the *nested function*. Nested functions are not an independent function type; they exist within the body of one of the other types of functions discussed here (with the exception of anonymous functions), and also within other nested functions.

One type of function that is not discussed in this chapter is the MATLAB built-in function. Built-ins are defined only as executables internal to MATLAB. See "Built-In Functions" on page 3-109 for more information.

Anonymous Functions

In this section...

"Constructing an Anonymous Function" on page 5-3

"Arrays of Anonymous Functions" on page 5-6

"Outputs from Anonymous Functions" on page 5-7

"Variables Used in the Expression" on page 5-8

"Examples of Anonymous Functions" on page 5-11

Constructing an Anonymous Function

Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function either at the MATLAB command line or in any M-file function or script.

The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Starting from the right of this syntax statement, the term `expr` represents the body of the function: the code that performs the main task your function is to accomplish. This consists of any single, valid MATLAB expression. Next is `arglist`, which is a comma-separated list of input arguments to be passed to the function. These two components are similar to the body and argument list components of any function.

Leading off the entire right side of this statement is an `@` sign. The `@` sign is the MATLAB operator that constructs a function handle. Creating a function handle for an anonymous function gives you a means of invoking the function. It is also useful when you want to pass your anonymous function in a call to some other function. The `@` sign is a required part of an anonymous function definition.

Note Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function. The constructor uses a different syntax: `fhandle = @functionname` (e.g., `fhandle = @sin`). To find out more about function handles, see "Function Handles" on page 4-22.

The syntax statement shown above constructs the anonymous function, returns a handle to this function, and stores the value of the handle in variable `fhandle`. You can use this function handle in the same way as any other MATLAB function handle.

Simple Example

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
   0.3333
```

A Two-Input Example

As another example, you could create the following anonymous function that uses two input arguments, `x` and `y`. (The example assumes that variables `A` and `B` are already defined):

```
sumAxBy = @(x, y) (A*x + B*y);

whos sumAxBy
Name        Size                    Bytes  Class

sumAxBy     1x1                        16  function_handle
```

To call this function, assigning `5` to `x` and `7` to `y`, type

```
sumAxBy(5, 7)
```

Evaluating With No Input Arguments

For anonymous functions that do not take any input arguments, construct the function using empty parentheses for the input argument list:

```
t = @() datestr(now);
```

Also use empty parentheses when invoking the function:

```
t()

ans =
04-Sep-2003 10:17:59
```

You must include the parentheses. If you type the function handle name with no parentheses, MATLAB just identifies the handle; it does not execute the related function:

```
t

t =
    @() datestr(now)
```

Arrays of Anonymous Functions

To store multiple anonymous functions in an array, use a cell array. The example shown here stores three simple anonymous functions in cell array A:

```
A = {@(x)x.^2, @(y)y+10, @(x,y)x.^2+y+10}
A =
    [@(x)x.^2]    [@(y)y+10]    [@(x,y)x.^2+y+10]
```

Execute the first two functions in the cell array by referring to them with the usual cell array syntax, A{1} and A{2}:

```
A{1}(4) + A{2}(7)
ans =
    33
```

Do the same with the third anonymous function that takes two input arguments:

```
A{3}(4, 7)
ans =
    33
```

Space Characters in Anonymous Function Elements

Note that while using space characters in the definition of any function can make your code easier to read, spaces in the body of an anonymous function that is defined in a cell array can sometimes be ambiguous to MATLAB. To ensure accurate interpretation of anonymous functions in cell arrays, you can do any of the following:

- Remove all spaces from at least the body (not necessarily the argument list) of each anonymous function:

  ```
  A = {@(x)x.^2, @(y)y+10, @(x, y)x.^2+y+10};
  ```

- Enclose in parentheses any anonymous functions that include spaces:

  ```
  A = {(@(x)x .^ 2), (@(y) y +10), (@(x, y) x.^2 + y+10)};
  ```

- Assign each anonymous function to a variable, and use these variable names in creating the cell array:

```
A1 = @(x)x .^ 2;  A2 = @(y) y +10;  A3 = @(x, y)x.^2 + y+10;
A = {A1, A2, A3};
```

Outputs from Anonymous Functions

As with other MATLAB functions, the number of outputs returned by an anonymous function depends mainly on how many variables you specify to the left of the equals (=) sign when you call the function.

For example, consider an anonymous function `getPersInfo` that returns a person's address, home phone, business phone, and date of birth, in that order. To get someone's address, you can call the function specifying just one output:

```
address = getPersInfo(name);
```

To get more information, specify more outputs:

```
[address, homePhone, busPhone] = getPersInfo(name);
```

Of course, you cannot specify more outputs than the maximum number generated by the function, which is four in this case.

Example

The anonymous `getXLSData` function shown here calls the MATLAB `xlsread` function with a preset spreadsheet filename (`records.xls`) and a variable worksheet name (`worksheet`):

```
getXLSData = @(worksheet) xlsread('records.xls', worksheet);
```

The `records.xls` worksheet used in this example contains both numeric and text data. The numeric data is taken from instrument readings, and the text data describes the category that each numeric reading belongs to.

Because the MATLAB `xlsread` function is defined to return up to three values (numeric, text, and raw data), `getXLSData` can also return this same number of values, depending on how many output variables you specify to the left of the equals sign in the call. Call `getXLSData` a first time, specifying only a single (numeric) output, `dNum`:

```
dNum = getXLSData('Week 12');
```

Display the data that is returned using a `for` loop. You have to use generic names (`v1`, `v2`, `v3`) for the categories, due to the fact that the text of the real category names was not returned in the call:

```
for k = 1:length(dNum)
   disp(sprintf('%s    v1: %2.2f    v2: %d    v3: %d', ...
      datestr(clock, 'HH:MM'), dNum(k,1), dNum(k,2), ...
      dNum(k,3)));
   end
```

Here is the output from the first call:

```
12:55   v1: 78.42    v2: 32    v3: 37
13:41   v1: 69.73    v2: 27    v3: 30
14:26   v1: 77.65    v2: 17    v3: 16
15:10   v1: 68.19    v2: 22    v3: 35
```

Now try this again, but this time specifying two outputs, numeric (`dNum`) and text (`dTxt`):

```
[dNum, dTxt] = getXLSData('Week 12');

for k = 1:length(dNum)
   disp(sprintf('%s    %s: %2.2f    %s: %d    %s: %d', ...
      datestr(clock, 'HH:MM'), dTxt{1}, dNum(k,1), ...
      dTxt{2}, dNum(k,2), dTxt{3}, dNum(k,3)));
   end
```

This time, you can display the category names returned from the spreadsheet:

```
12:55    Temp: 78.42    HeatIndex: 32    WindChill: 37
13:41    Temp: 69.73    HeatIndex: 27    WindChill: 30
14:26    Temp: 77.65    HeatIndex: 17    WindChill: 16
15:10    Temp: 68.19    HeatIndex: 22    WindChill: 35
```

Variables Used in the Expression

Anonymous functions commonly include two types of variables:

- Variables specified in the argument list. These often vary with each function call.

- Variables specified in the body of the expression. MATLAB captures these variables and holds them constant throughout the lifetime of the function handle.

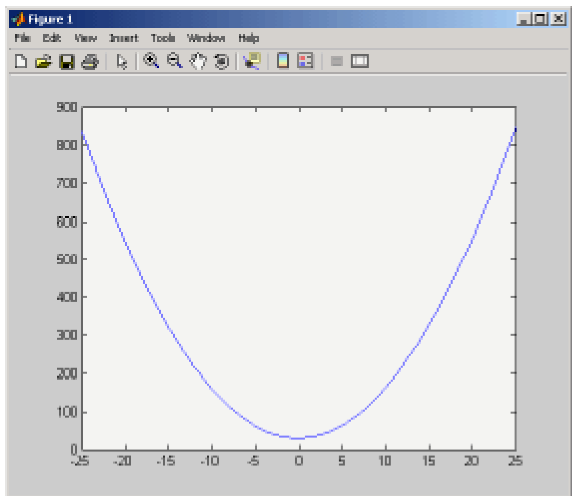
The latter variables must have a value assigned to them at the time you construct an anonymous function that uses them. Upon construction, MATLAB captures the current value for each variable specified in the body of that function. The function will continue to associate this value with the variable even if the value should change in the workspace or go out of scope.

The fact that MATLAB captures the values of these variables when the handle to the anonymous function is constructed enables you to execute an anonymous function from anywhere in the MATLAB environment, even outside the scope in which its variables were originally defined. But it also means that to supply new values for any variables specified within the expression, you must reconstruct the function handle.

Changing Variables Used in an Anonymous Function

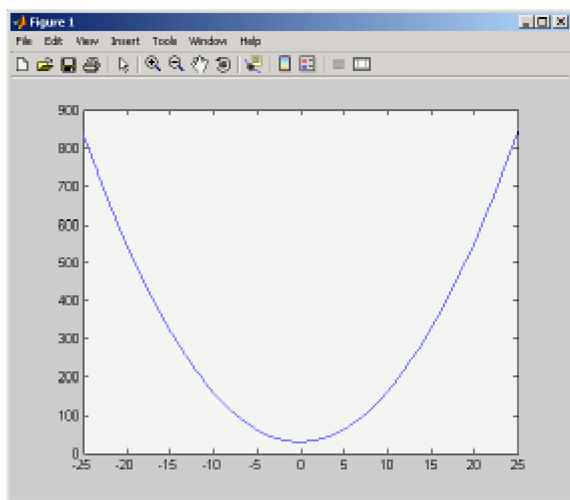
The second statement shown below constructs a function handle for an anonymous function called `parabola` that uses variables `a`, `b`, and `c` in the expression. Passing the function handle to the MATLAB `fplot` function plots it out using the initial values for these variables:

```
a = 1.3;   b = .2;   c = 30;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



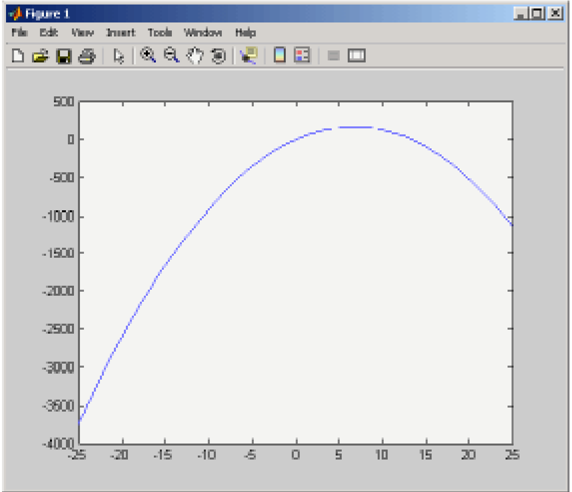
If you change the three variables in the workspace and replot the figure, the parabola remains unchanged because the parabola function is still using the initial values of a, b, and c:

```
a = -3.9;   b = 52;   c = 0;
fplot(parabola, [-25 25])
```



To get the function to use the new values, you need to reconstruct the function handle, causing MATLAB to capture the updated variables. Replot using the new construct, and this time the parabola takes on the new values:

```
a = -3.9;   b = 52;   c = 0;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



For the purposes of this example, there is no need to store the handle to the anonymous function in a variable (`parabola`, in this case). You can just construct and pass the handle right within the call to `fplot`. In this way, you update the values of `a`, `b`, and `c` on each call:

```
fplot(@(x) a*x.^2 + b*x + c, [-25 25])
```

Examples of Anonymous Functions

This section shows a few examples of how you can use anonymous functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

- "Example 1 — Passing a Function to quad" on page 5-12
- "Example 2 — Multiple Anonymous Functions" on page 5-13

Example 1 — Passing a Function to quad

The equation shown here has one variable t that can vary each time you call the function, and two additional variables, g and omega. Leaving these two variables flexible allows you to avoid having to hardcode values for them in the function definition:

```
x = g * cos(omega * t)
```

One way to program this equation is to write an M-file function, and then create a function handle for it so that you can pass the function to other functions, such as the MATLAB quad function as shown here. However, this requires creating and maintaining a new M-file for a purpose that is likely to be temporary, using a more complex calling syntax when calling quad, and passing the g and omega parameters on every call. Here is the function M-file:

```
function f = vOut(t, g, omega)
f = g * cos(omega * t);
```

This code has to specify g and omega on each call:

```
g = 2.5;  omega = 10;

quad(@vOut, 0, 7, [], [], g, omega)
ans =
   0.1935

quad(@vOut, -5, 5, [], [], g, omega)
ans =
   -0.1312
```

You can simplify this procedure by setting the values for g and omega just once at the start, constructing a function handle to an anonymous function that only lasts the duration of your MATLAB session, and using a simpler syntax when calling quad:

```
g = 2.5;  omega = 10;
```

```
quad(@(t) (g * cos(omega * t)), 0, 7)
ans =
   0.1935

quad(@(t) (g * cos(omega * t)), -5, 5)
ans =
   -0.1312
```

To preserve an anonymous function from one MATLAB session to the next, save the function handle to a MAT-file

```
save anon.mat f
```

and then load it into the MATLAB workspace in a later session:

```
load anon.mat f
```

Example 2 — Multiple Anonymous Functions

This example solves the following equation by combining two anonymous functions:

$$g(c) = \int_0^1 (x^2 + cx + 1)dx$$

The equivalent anonymous function for this expression is

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));
```

This was derived as follows. Take the parenthesized part of the equation (the integrand) and write it as an anonymous function. You don't need to assign the output to a variable as it will only be passed as input to the quad function:

```
@(x) (x.^2 + c*x + 1)
```

Next, evaluate this function from zero to one by passing the function handle, shown here as the entire anonymous function, to quad:

```
quad(@(x) (x.^2 + c*x + 1), 0, 1)
```

Supply the value for c by constructing an anonymous function for the entire equation and you are done:

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));

g(2)
ans =
   2.3333
```

Primary M-File Functions

The first function in any M-file is called the *primary function*. Following the primary function can be any number of subfunctions, which can serve as subroutines to the primary function.

Under most circumstances, the primary function is the only function in an M-file that you can call from the MATLAB command line or from another M-file function. You invoke this function using the name of the M-file in which it is defined.

For example, the `average` function shown here resides in the file `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.

y = sum(x)/length(x);      % Actual computation
```

You can invoke this function from the MATLAB command line with this command to find the average of three numbers:

```
average([12 60 42])
```

Note that it is customary to give the primary function the same name as the M-file in which it resides. If the function name differs from the filename, then you must use the filename to invoke the function.

Nested Functions

In this section...

"Writing Nested Functions" on page 5-16

"Calling Nested Functions" on page 5-17

"Variable Scope in Nested Functions" on page 5-19

"Using Function Handles with Nested Functions" on page 5-21

"Restrictions on Assigning to Variables" on page 5-26

"Examples of Nested Functions" on page 5-27

Writing Nested Functions

You can define one or more functions within another function in MATLAB. These inner functions are said to be *nested* within the function that contains them. You can also nest functions within other nested functions.

To write a nested function, simply define one function within the body of another function in an M-file. Like any M-file function, a nested function contains any or all of the components described in "Basic Parts of an M-File" on page 4-8. In addition, you must always terminate a nested function with an `end` statement:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end
...
end
```

Note M-file functions don't normally require a terminating `end` statement. This rule does not hold, however, when you nest functions. If an M-file contains one or more nested functions, you must terminate *all* functions (including subfunctions) in the M-file with `end`, whether or not they contain nested functions.

Example — More Than One Nested Function

This example shows function A and two additional functions nested inside A at the same level:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end

   function z = C(p4)
   ...
   end
...
end
```

Example — Multiply Nested Functions

This example shows multiply nested functions, C nested inside B, and B in A:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
      function z = C(p4)
      ...
      end
   ...
   end
...
end
```

Calling Nested Functions

You can call a nested function

- From the level immediately above it. (In the following code, function A can call B or D, but not C or E.)
- From a function nested at the same level within the same parent function. (Function B can call D, and D can call B.)

- From a function at any lower level. (Function C can call B or D, but not E.)

```
function A(x, y)              % Primary function
B(x, y);
D(y);

   function B(x, y)           % Nested in A
   C(x);
   D(y);

      function C(x)           % Nested in B
      D(x);
      end
   end

   function D(x)              % Nested in A
   E(x);

      function E(x)           % Nested in D
      ...
      end
   end
end
```

You can also call a subfunction from any nested function in the same M-file.

You can pass variable numbers of arguments to and from nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances. See "Passing Optional Arguments to Nested Functions" in the MATLAB Programming documentation for more information on this.

Note If you construct a function handle for a nested function, you can call the nested function from any MATLAB function that has access to the handle. See “Using Function Handles with Nested Functions” on page 5-21.

Variable Scope in Nested Functions

The scope of a variable is the range of functions that have direct access to the variable to set, modify, or acquire its value. When you define a local (i.e., nonglobal) variable within a function, its scope is normally restricted to that function alone. For example, subfunctions do not share variables with the primary function or with other subfunctions. This is because each function and subfunction stores its variables in its own separate workspace.

Like other functions, a nested function has its own workspace. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

In the following two examples, variable `x` is stored in the workspace of the outer `varScope` function and can be read or written to by all functions nested within it.

```
function varScope1
x = 5;
nestfun1
   function nestfun1
      nestfun2

      function nestfun2
         x = x + 1
      end
   end
end
```

```
function varScope2
nestfun1
   function nestfun1
      nestfun2

      function nestfun2
         x = 5;
      end
   end
x = x + 1
end
```

As a rule, a variable used or defined within a nested function resides in the workspace of the outermost function that both contains the nested function and accesses that variable. The scope of this variable is then the function to which this workspace belongs, and all functions nested to any level within that function.

In the next example, the outer function, `varScope3`, does not access variable `x`. Following the rule just stated, `x` is unknown to the outer function and thus is

not shared between the two nested functions. In fact, there are two separate `x` variables in this example: one in the function workspace of `nestfun1` and one in the function workspace of `nestfun2`. When `nestfun2` attempts to update `x`, it fails because `x` does not yet exist in this workspace:

```
function varScope3
nestfun1
nestfun2

   function nestfun1
      x = 5;
   end

   function nestfun2
      x = x + 1
   end
end
```

The Scope of Output Variables

Variables containing values returned by a nested function are not in the scope of outer functions. In the two examples shown here, the one on the left fails in the second to last line because, although the *value* of `y` is returned by the nested function, the *variable* `y` is local to the nested function, and unknown to the outer function. The example on the right assigns the return value to a variable, `z`, and then displays the value of `z` correctly.

Incorrect	Correct

<pre>function varScope4
x = 5; nestfun;
   function y = nestfun
      y = x + 1;
   end
 	
y
end</pre>
	<pre>function varScope5
	x = 5;
	z = nestfun;
	   function y = nestfun
	      y = x + 1;
	   end
	
	z
	end</pre>

Using Function Handles with Nested Functions

Every function has a certain *scope*, that is, a certain range of other functions to which it is visible. A function's scope determines which other functions can call it. You can call a function that is out of scope by providing an alternative means of access to it in the form of a function handle. (The function handle, however, must be within the scope of its related function when you construct the handle.) Any function that has access to a function handle can call the function with which the handle is associated.

Note Although you can call an out of scope function by means of a function handle, the handle itself must be within the scope of its related function at the time it is constructed.

The section on "Calling Nested Functions" on page 5-17 defines the scope of a nested function. As with other types of functions, you can make a nested function visible beyond its normal scope with a function handle. The following function `getCubeHandle` constructs a handle for nested function `findCube` and returns its handle, `h`, to the caller. The `@` sign placed before a function name (e.g., `@findCube`) is the MATLAB operator that constructs a handle for that function:

```
function h = getCubeHandle
h = @findCube;                  % Function handle constructor

   function cube = findCube(X)   % Nested function
      cube = X .^ 3;
   end
end
```

Call `getCubeHandle` to obtain the function handle to the nested function `findCube`. Assign the function handle value returned by `getCubeHandle` to an output variable, `cubeIt` in this case:

```
cubeIt = getCubeHandle;
```

You can now use this variable as a means of calling `findCube` from outside of its M-file:

```
cubeIt(8)
ans =
    512
```

Note When calling a function by means of its handle, use the same syntax as if you were calling a function directly. But instead of calling the function by its name (e.g., `strcmp(S1, S2)`), use the variable that holds the function handle (e.g., `fhandle(S1, S2)`).

Function Handles and Nested Function Variables

One characteristic of nested functions that makes them different from other MATLAB functions is that they can share nonglobal variables with certain other functions within the same M-file. A nested function `nFun` can share variables with any outer function that contains `nFun`, and with any function nested within `nFun`. This characteristic has an impact on how certain variables are stored when you construct a handle for a nested function.

Defining Variables When Calling Via Function Handle. The example below shows a primary function `getHandle` that returns a function handle for the nested function `nestFun`. The `nestFun` function uses three different types of variables. The `VLoc` variable is local to the nested function, `VInp` is passed in when the nested function is called, and `VExt` is defined by the outer function:

```
function h = getHandle(X)
h = @nestFun;
VExt = someFun(X);

   function nestFun(VInp)
   VLoc = 173.5;
   doSomeTask(VInp, VLoc, VExt);
   end
end
```

As with any function, when you call `nestFun`, you must ensure that you supply the values for any variables it uses. This is a straightforward matter

when calling the nested function directly (that is, calling it from `getHandle`). `VLoc` has a value assigned to it within `nestFun`, `VInp` has its value passed in, and `VExt` acquires its value from the workspace it shares with `getHandle`.

However, when you call `nestFun` using a function handle, only the nested function executes; the outer function, `getHandle`, does not. It might seem at first that the variable `VExt`, otherwise given a value by `getHandle`, has no value assigned to it in the case. What in fact happens though is that MATLAB stores variables such as `VExt` inside the function handle itself when it is being constructed. These variables are available for as long as the handle exists.

The `VExt` variable in this example is considered to be *externally scoped* with respect to the nested function. Externally scoped variables that are used in nested functions for which a function handle exists are stored within the function handle. So, function handles not only contain information about accessing a function. For nested functions, a function handle also stores the values of any externally scoped variables required to execute the function.

Example Using Externally Scoped Variables

The `sCountFun` and `nCountFun` functions shown below return function handles for subfunction `subCount` and nested function `nestCount`, respectively.

These two inner functions store a persistent value in memory (the value is retained in memory between function calls), and then increment this value on every subsequent call. `subCount` makes its `count` value persistent with an explicit `persistent` declaration. In `nestCount`, the count variable is externally scoped and thus is maintained in the function handle:

<table>
<tr><th>Using a Subfunction</th><th>Using a Nested Function</th></tr>
<tr><td><pre>function h = sCountFun(X)
h = @subCount;
count = X
subCount(0, count);
function subCount(incr, ini)
persistent count;
initializing = nargin > 1;
if initializing
   count = ini; else
   count = count + incr
end</pre></td><td><pre>function h = nCountFun(X)
h = @nestCount;
count = X
   function nestCount(incr)
      count = count + incr
   end
end</pre></td></tr>
</table>

When `sCountFun` executes, it passes the initial value for `count` to the `subCount` subfunction. Keep in mind that the `count` variable in `sCountFun` is not the same as the count variable in `subCount`; they are entirely independent of each other. Whenever `subCount` is called via its function handle, the value for `count` comes from its persistent place in memory.

In `nestCount`, the `count` variable again gets its value from the primary function when called from within the M-file. However, in this case the `count` variable in the primary and nested functions are one and the same. When `nestCount` is called by means of its function handle, the value for `count` is assigned from its storage within the function handle.

Running the Example. The `subCount` and `nestCount` functions increment a value in memory by another value that you pass as an input argument. Both of these functions give the same results.

Get the function handle to `nestCount`, and initialize the `count` value to a four-element vector:

```
h = nCountFun([100 200 300 400])
count =
   100   200   300   400
```

Increment the persistent vector by 25, and then by 42:

```
h(25)
```

```
count =
   125   225   325   425

h(42)
count =
   167   267   367   467
```

Now do the same using `sCountFun` and `subCount`, and verify that the results are the same.

Note If you construct a new function handle to `subCount` or `nestCount`, the former value for `count` is no longer retained in memory. It is replaced by the new value.

Separate Instances of Externally Scoped Variables

The code shown below constructs two separate function handles to the same nested function, `nestCount`, that was used in the last example. It assigns the handles to fields `counter1` and `counter2` of structure `s`. These handles reference different instances of the `nestCount` function. Each handle also maintains its own separate value for the externally scoped `count` variable.

Call `nCountFun` twice to get two separate function handles to `nestCount`. Initialize the two instances of `count` to two different vectors:

```
s.counter1 = nCountFun([100 200 300 400]);
count =
   100   200   300   400

s.counter2 = nCountFun([-100 -200 -300 -400]);
count =
  -100  -200  -300  -400
```

Now call `nestCount` by means of each function handle to demonstrate that MATLAB increments the two `count` variables individually.

Increment the first counter:

```
s.counter1(25)
```

```
count =
   125   225   325   425
s.counter1(25)
count =
   150   250   350   450
```

Now increment the second counter:

```
s.counter2(25)
count =
   -75  -175  -275  -375
s.counter2(25)
count =
   -50  -150  -250  -350
```

Go back to the first counter and you can see that it keeps its own value for `count`:

```
s.counter1(25)
count =
   175   275   375   475
```

Restrictions on Assigning to Variables

The scoping rules for nested, and in some cases anonymous, functions require that all variables used within the function be present in the text of the M-file code. Adding variables to the workspace of this type of function at run time is not allowed.

MATLAB issues an error if you attempt to dynamically add a variable to the workspace of an anonymous function, a nested function, or a function that contains a nested function. Examples of operations that might use dynamic assignment in this way are shown in the table below.

Type of Operation	How to Avoid Using Dynamic Assignment
Evaluating an expression using `eval` or `evalin`, or assigning a variable with `assignin`	As a general suggestion, it is best to avoid using the `eval`, `evalin`, and `assignin` functions altogether.
Loading variables from a MAT-file with the `load` function	Use the form of `load` that returns a MATLAB structure.
Assigning to a variable in a MATLAB script	Convert the script to a function, where argument- and result-passing can often clarify the code as well.
Assigning to a variable in the MATLAB debugger	You can declare the variable to be `global`. For example, to create a variable X for temporary use in debugging, use `K>> global X;    X = value`

One way to avoid this error in the other cases is to pre-declare the variable in the desired function.

Examples of Nested Functions

This section shows a few examples of how you can use nested functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

- "Example 1 — Creating a Function Handle for a Nested Function" on page 5-27
- "Example 2 — Function-Generating Functions" on page 5-29

Example 1 — Creating a Function Handle for a Nested Function

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB `fplot` function to plot the parabola

shape. The `makeParabola` function shown here constructs and returns a function handle `fhandle` for the nested `parabola` function. This handle gets passed to `fplot`:

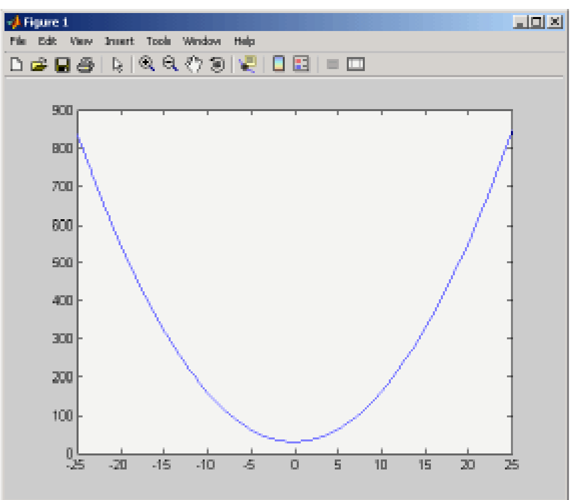
```
function fhandle = makeParabola(a, b, c)
% MAKEPARABOLA returns a function handle with parabola
% coefficients.

fhandle = @parabola;     % @ is the function handle constructor

   function y = parabola(x)
   y = a*x.^2 + b*x + c;
   end
end
```

Assign the function handle returned from the call to a variable (`h`) and evaluate the function at points 0 and 25:

```
h = makeParabola(1.3, .2, 30)
h =
    @makeParabola/parabola

h(0)
ans =
    30

h(25)
ans =
  847.5000
```

Now pass the function handle `h` to the `fplot` function, evaluating the parabolic equation from `x = -25` to `x = +25`:

```
fplot(h, [-25 25])
```



Example 2 — Function-Generating Functions

The fact that a function handle separately maintains a unique instance of the function from which it is constructed means that you can generate multiple handles for a function, each operating independently from the others. The function in this example makes IIR filtering functions by constructing function handles from nested functions. Each of these handles maintains its own internal state independent of the others.

The function `makeFilter` takes IIR filter coefficient vectors `a` and `b` and returns a filtering function in the form of a function handle. Each time a new input value x_n is available, you can call the filtering function to get the new output value y_n. Each filtering function created by `makeFilter` keeps its own private `a` and `b` vectors, in addition to its own private state vector, in the form of a transposed direct form II delay line:

```
function [filtfcn, statefcn] = makeFilter(b, a)
%   FILTFCN = MAKEFILTER(B, A) creates an IIR filtering
%   function and returns it in the form of a function handle,
```

```
%   FILTFCN. Each time you call FILTFCN with a new filter
%   input value, it computes the corresponding new filter
%   output value, updating its internal state vector at the
%   same time.
%
%   [FILTFCN, STATEFCN] = MAKEFILTER(B, A) also returns a
%   function (in the form of a function handle, STATEFCN)
%   that can return the filter's internal state.  The internal
%   state vector is in the form of a transposed direct form
%   II delay line.

%   Initialize state vector. To keep this example a bit
%   simpler, assume that a and b have the same length.
%   Also assume that a(1) is 1.

v = zeros(size(a));

filtfcn =  @iirFilter;
statefcn = @getState;

   function yn = iirFilter(xn)
      % Update the state vector
      v(1) = v(2) + b(1) * xn;
      v(2:end-1) = v(3:end) + b(2:end-1) * xn - ...
         a(2:end-1) * v(1);
      v(end) = b(end) * xn - a(end) * v(1);

      % Output is the first element of the state vector.
      yn = v(1);
   end

   function vOut = getState
      vOut = v;
   end
end
```

This sample session shows how `makeFilter` works. Make a filter that has a decaying exponential impulse response and then call it a few times in succession to see the output values change:

```
[filt1, state1] = makeFilter([1 0], [1 -.5]);

% First input to the filter is 1.
filt1(1)
ans =
   1

% Second input to the filter is 0.
filt1(0)
ans =
   0.5000

filt1(0)
ans =
   0.2500

% Show the filter's internal state.
state1()
ans =
   0.2500    0.1250

% Hit the filter with another impulse.
filt1(1)
ans =
   1.1250

% How did the state change?
state1()
ans =
   1.1250    0.5625

% Make an averaging filter.
filt2 = makeFilter([1 1 1]/3, [1 0 0]);

% Put a step input into filt2.
filt2(1)
ans =
   0.3333

filt2(1)
```

```
ans =
   0.6667

filt2(1)
ans =
   1

% The two filter functions can be used independently.
filt1(0)
ans =
   0.5625
```

As an extension of this example, suppose you were looking for a way to develop simulations of different filtering structures and compare them. This might be useful if you were interested in obtaining the range of values taken on by elements of the state vector, and how those values compare with a different filter structure. Here is one way you could capture the filter state at each step and save it for later analysis:

Call `makeFilter` with inputs `v1` and `v2` to construct function handles to the `iirFilter` and `getState` subfunctions:

```
[filtfcn, statefcn] = makeFilter(v1, v2);
```

Call the `iirFilter` and `getState` functions by means of their handles, passing in random values:

```
x = rand(1, 20);
for k = 1:20
    y(k) = filtfcn(x(k));
    states{k} = statefcn();  % Save the state at each step.
end
```

Subfunctions

In this section...

"Overview" on page 5-33
"Calling Subfunctions" on page 5-34
"Accessing Help for a Subfunction" on page 5-34

Overview

M-files can contain code for more than one function. Additional functions within the file are called *subfunctions*, and these are only visible to the primary function or to other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first:

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u, n);
med = median(u, n);

function a = mean(v, n)            % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n)          % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
   m = w((n+1) / 2);
else
   m = (w(n/2) + w(n/2+1)) / 2;
end
```

The subfunctions `mean` and `median` calculate the average and median of the input list. The primary function `newstats` determines the length of the list and calls the subfunctions, passing to them the list length `n`.

Subfunctions cannot access variables used by other subfunctions, even within the same M-file, or variables used by the primary function of that M-file, unless you declare them as global within the pertinent functions, or pass them as arguments.

Calling Subfunctions

When you call a function from within an M-file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard M-file or built-in function on your search path. Because it checks for a subfunction first, you can override existing M-files using subfunctions with the same name.

Accessing Help for a Subfunction

You can write help for subfunctions using the same rules that apply to primary functions. To display the help for a subfunction, precede the subfunction name with the name of the M-file that contains the subfunction (minus file extension) and a `>` character.

For example, to get help on subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

Private Functions

In this section...
"Overview" on page 5-35
"Private Directories" on page 5-35
"Accessing Help for a Private Function" on page 5-36

Overview

Private functions are functions that reside in subdirectories with the special name `private`. These functions are called *private* because they are visible only to M-file functions and M-file scripts that meet these conditions:

- A function that calls a private function must be defined in an M-file that resides in the directory immediately above that `private` subdirectory.
- A script that calls a private function must itself be called from an M-file function that has access to the private function according to the above rule.

For example, assume the directory `newmath` is on the MATLAB search path. A subdirectory of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a nonprivate M-file named `test.m`.

Primary functions and subfunctions can also be implemented as private functions.

Private Directories

You can create your own private directories simply by creating subdirectories called `private` using the standard procedures for creating directories or folders on your computer. Do not place these private directories on your path.

Accessing Help for a Private Function

You can write help for private functions using the same rules that apply to primary functions. To display the help for a private function, precede the private function name with `private/`.

For example, to get help on private function `myprivfun`, type

```
help private/myprivfun
```

Overloaded Functions

Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers.

MATLAB overloaded functions reside in subdirectories having a name starting with the symbol `@` and followed by the name of a recognized MATLAB data type. For example, functions in the `\@double` directory execute when invoked with arguments of MATLAB type `double`. Those in an `\@int32` directory execute when invoked with arguments of MATLAB type `int32`.

See "Classes and Objects: An Overview" on page 9-2 for more information on overloading functions in MATLAB.

Data Import and Export

Overview (p. 6-3)	See what MATLAB offers in the way of import and export facilities for various data formats.
Supported File Formats (p. 6-9)	View the list of file formats and file extensions supported for MATLAB import and export along with the functions used with each type.
Using the Import Wizard (p. 6-11)	Import many types of binary data using this GUI-based interface.
Accessing Files with Memory-Mapping (p. 6-23)	Get faster, more efficient file I/O for very large files by accessing files on disk via pointers in memory.
Exporting Data to MAT-Files (p. 6-64)	Save data from your MATLAB session in a MAT-file, a binary data file designed specifically for MATLAB data.
Importing Data From MAT-Files (p. 6-72)	Load data that was saved to a MAT-file back into your MATLAB session.
Importing Text Data (p. 6-75)	Import ASCII text data into MATLAB using the Import Wizard and import functions.
Exporting Text Data (p. 6-84)	Export ASCII text data to MAT-files.
Working with Graphics Files (p. 6-90)	Import and export images stored in many different types of graphics files.

Working with Audio and Video Data (p. 6-93)	Import and export audio and video data.
Working with Spreadsheets (p. 6-98)	Interact with Microsoft Excel and Lotus 123 spreadsheets.
Using Low-Level File I/O Functions (p. 6-104)	Use the MATLAB low-level file I/O functions, such as fopen, fread, and fwrite.
Exchanging Files over the Internet (p. 6-117)	Exchange files over the Internet with MATLAB URL, zip, and e-mail functions.

Overview

In this section...

"File Types Supported by MATLAB" on page 6-3

"Other MATLAB I/O Capabilities" on page 6-5

"Functions Used in File Management" on page 6-7

For more information and examples on importing and exporting data, see Technical Note 1602:

`http://www.mathworks.com/support/tech-notes/1600/1602.html`

File Types Supported by MATLAB

MATLAB provides many ways to load data from disk files or the clipboard into the workspace, a process called *importing* data. Also there are many ways to save workspace variables to a disk file, a process called *exporting* data. Your choice of which import or export mechanism to use depends mainly on the format of the data being transferred: text, binary, or a standard format such as JPEG.

Note For unsupported high-level function data formats, you can use the MATLAB low-level file I/O functions if you know how the binary data is formatted in the file. See "Using Low-Level File I/O Functions" on page 6-104 for more information.

MATLAB has built-in capabilities to import and export the following types of data files:

- "Binary Data from a MATLAB Session" on page 6-4
- "Text Data" on page 6-4
- "Graphics Files" on page 6-4
- "Audio and Audio/Video Data" on page 6-4
- "Spreadsheets" on page 6-5

- "Data from the System Clipboard" on page 6-5
- "Information from the Internet" on page 6-5

Binary Data from a MATLAB Session

Using the MATLAB `save` and `load` functions, you can store all or part of the data in your MATLAB workspaces to disk, and then read that data back into MATLAB at a later time.

Text Data

In text format, the data values are American Standard Code for Information Interchange (ASCII) codes that represent alphabetic and numeric characters. ASCII text data can be viewed in a text editor. For more information about working with text data in MATLAB, see

- "Importing Text Data" on page 6-75
- "Exporting Text Data" on page 6-84

These sections also describe how to import and export to XML documents.

Graphics Files

MATLAB imports and exports images from many standard graphics file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats.

Audio and Audio/Video Data

MATLAB provides functions to enable you to interact with the following types of audio and audio/video files:

- NeXT/SUN SPARCstation sound
- Microsoft WAVE sound
- Audio/Video Interleaved (AVI)
- Windows-compatible sound devices
- Audio player and recorder objects

- Linear audio signals

Spreadsheets

You can use MATLAB to import and export data to the following types of spreadsheets:

- Microsoft Excel spreadsheets
- Lotus 123 spreadsheets

Data from the System Clipboard

Using the Import Wizard or the `clipboard` function, you can temporarily hold string data on your system's clipboard, and then paste it back into MATLAB.

Information from the Internet

From your MATLAB session, you can

- Send e-mail
- Download from the Internet
- Compress (zip) and decompress (unzip) files
- Connect to an FTP server to perform remote file operations

Other MATLAB I/O Capabilities

- "Using the Import Wizard" on page 6-5
- "Mapping Files to Memory" on page 6-6
- "Reading Files with Large Data Sets" on page 6-6
- "Low-Level File I/O" on page 6-6
- "Importing Data with Toolboxes" on page 6-7

Using the Import Wizard

The Import Wizard is a graphical user interface that simplified the process of locating and loading various types of data files into MATLAB. You do not need to know the format of the data to use this tool. You simply specify the

file that contains the data and the Import Wizard processes the file contents automatically. See the section on “Using the Import Wizard” on page 6-11.

Mapping Files to Memory

Memory—mapping enables you to read and write data in a file as if were stored in the computer’s dynamic memory. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file. See the section on “Accessing Files with Memory-Mapping” on page 6-23.

Reading Files with Large Data Sets

An efficient way to read files with large data sets is to read the file in segments and process the data as you go. This method requires significantly less memory than if you were to try reading in the entire file at once. Using the `textscan` function, you can read a specified amount of data from a file, and maintain a pointer to the location in the file where your last read operation ended and your next read is to begin.

This example opens a large data file and reads the file a segment at a time in a `for` loop. The code calls `textscan` to read a particular pattern of data (as specified by `format`) 10,000 times for each segment. Following each read, the subfunction `process_data` processes the data collected in cell array `segarray`:

```
format = '%s %n %s %8.2f %8.2f %8.2f %8.2f %u8';
file_id = fopen('largefile.dat', 'r');

for k = 1:segcount
   segarray = textscan(file_id, format, 10000);
   process_data(segarray);
end

fclose(file_id);
```

Low-Level File I/O

MATLAB also supports C-style, low-level I/O functions that you can use with any data format. For more information, see “Using Low-Level File I/O Functions” on page 6-104.

Importing Data with Toolboxes

In addition to MATLAB import functions, you can perform specialized import features using toolboxes. For example, use Database Toolbox for importing data from relational databases. Refer to the documentation on the specific toolbox to see what import features are offered.

Functions Used in File Management

The following functions are available in MATLAB to help you to create, manage, and locate the files and directories you work with. For more information on these and other file management functions, see "File Management Operations" in the Desktop Tools and Development Environment documentation:

Function	Description
`cd`	Switch your current working directory to another directory
`clipboard`	Copy and paste strings to and from the system clipboard
`copyfile`	Copy a file or directory to another location
`delete`	Delete the specified files
`dir`	List the files that reside in the specified directory
`edit`	Create a new M-file or edit an existing one
`exist`	Check the existence of a file or directory
`fileattrib`	Set or get attributes of a file or directory
`filebrowser`	Start the Current Directory Browser
`fileparts`	Show the components of a file name and its place on the path
`fullfile`	Build a full file name from its components
`ls`	List the contents of a specific directory
`mkdir`	Create a new directory
`movefile`	Move a file or directory to a new location
`open`	Open files based on extension
`pwd`	Identify the directory you are currently working in

Function	Description
`recycle`	Set an option to move deleted files to recycle folder
`rmdir`	Delete a specific directory
`what`	List the MATLAB files in a specific directory
`which`	Locate functions and files

Supported File Formats

The table below shows the file formats that you can read or write from MATLAB along with the functions that support each format.

File Format	File Content	Extension	Functions
MATLAB formatted	Saved MATLAB workspace	.mat	load, save
Text	Text	any	textscan
	Text	any	textread
	Delimited text	any	dlmread, dlmwrite
	Comma-separated numbers	.csv	csvread, csvwrite
Extended Markup Language	XML-formatted text	.xml	xmlread, xmlwrite
Audio	NeXT/SUN sound	.au	auread, auwrite
	Microsoft WAVE sound	.wav	wavread, wavwrite
Movie	Audio/video	.avi	aviread
Scientific data	Data in Common Data Format	.cdf	cdfread, cdfwrite
	Flexible Image Transport System data	.fits	fitsread
	Data in Hierarchical Data Format	.hdf	hdfread
Spreadsheet	Excel worksheet	.xls	xlsread, xlswrite
	Lotus 123 worksheet	.wk1	wk1read, wk1write

File Format	File Content	Extension	Functions
Graphics	TIFF image	`.tiff`	`imread`, `imwrite`
	PNG image	`.png`	same
	HDF image	`.hdf`	same
	BMP image	`.bmp`	same
	JPEG image	`.jpeg`	same
	GIF image	`.gif`	same
	PCX image	`.pcx`	same
	XWD image	`.xwd`	same
	Cursor image	`.cur`	same
	Icon image	`.ico`	same

Using the Import Wizard

In this section...

"Overview" on page 6-11

"Starting the Import Wizard" on page 6-11

"Previewing Contents of the File or Clipboard [Text only]" on page 6-13

"Specifying Delimiters and Header Format [Text only]" on page 6-14

"Determining Assignment to Variables" on page 6-15

"Automated M-Code Generation" on page 6-18

"Writing Data to the Workspace" on page 6-21

Overview

The easiest way to import data into MATLAB is to use the Import Wizard. You do not need to know the format of the data to use this tool. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically. You can also use the Import Wizard to import HDF data. See "Using the HDF Import Tool" on page 7-36"Using the HDF Import Tool" on page 7-36 for more information.

The sections on Previewing Contents of the File or Clipboard and Specifying Delimiters and Header Format apply only to text files and the clipboard.

Starting the Import Wizard

To start the Import Wizard and select the source to import, see

- "Importing from a File" on page 6-12
- "Importing from the Clipboard" on page 6-12

If you use the `uiimport` function to start the Wizard, you can choose to have the imported data written to a MATLAB structure. See "Importing to a Structure" on page 6-12.

Importing from a File

To start the Wizard and use a file browser to locate the file to import, use one of the menu options or MATLAB commands shown here:

- Select **Import Data** from the **File** menu
- Type `uiimport -file`
- Type `uiimport`, and then click **Browse**

If you already know the name of the file to import, use one of the following means to initiate the operation:

- In the Current Directory browser, right-click the filename and select **Import Data**
- Type `uiimport filename`, where `filename` is an unquoted string containing the name of the file to import.

Importing from the Clipboard

To import from the system clipboard, use one of the menu options or MATLAB commands shown here:

- Select **Paste to Workspace** from the **Edit** menu
- Type `uiimport -pastespecial`
- Type `uiimport`, and then click **Clipboard**

Importing to a Structure

Specifying an output argument with the `uiimport` command tells MATLAB to return the imported data in the fields of a single structure rather than as separate variables.

The command

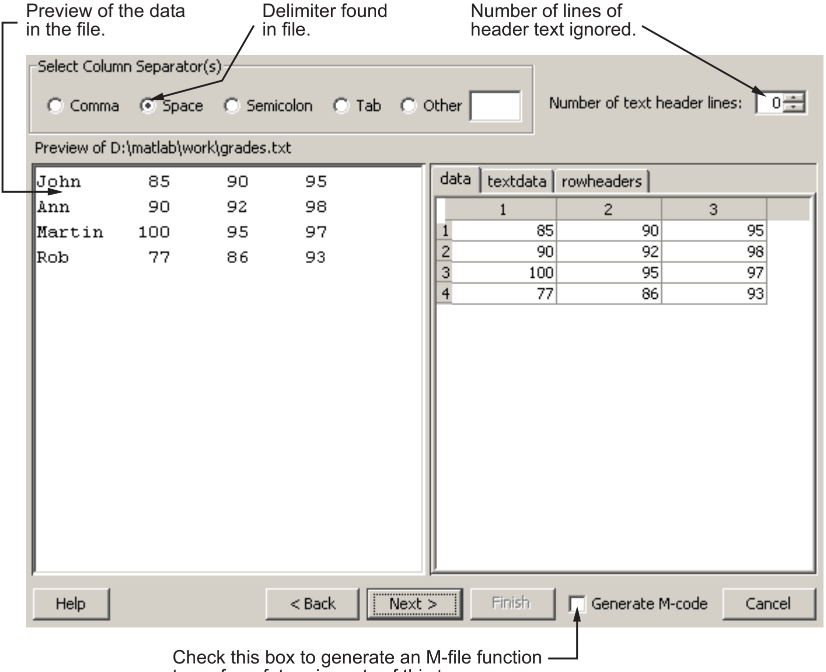
```
S = uiimport('filename')
```

imports the file `filename` to the fields of structure `S`. The `filename` argument is a single-quoted string containing the name of the file to import.

If you are importing from a binary file, skip ahead to step 4: Determine Assignment to Variables.

Previewing Contents of the File or Clipboard [Text only]

When the Import Wizard imports text data from a file or the clipboard, it opens the dialog box shown here and displays a portion of the raw data in the preview pane on the left. You can use this display to verify that the file contains the data you expect.



The pane on the right side of the dialog box shows how MATLAB has assigned the imported data to a default set of variables. The variable names appear in the tabs above the display pane. Click any of these tabs to see the values assigned to that variable. The variable names are derived from categories into which the Import Wizard has sorted the data. These are

- `rowheaders`—Column vector containing the names of all row headers.
- `colheaders`—Row vector containing the names of all column headers.
- `textdata`—Matrix containing all imported text data. Empty elements are set to `''`.
- `data`—Matrix containing all imported numeric data. Empty elements are set to `NaN`.

If the imported file or clipboard contains *only* numeric or *only* text data, then the Import Wizard does not use the variable names shown above. Instead, it assigns all of the data to just one variable:

- For data imported from a text file, the name of the variable is the same as the filename, minus the file extension.
- For data imported from the clipboard, the name of the variable is `A_pastespecial`.

Specifying Delimiters and Header Format [Text only]

Using the options shown at the top of the Import Wizard dialog box, you can specify a delimiter character for separating data items, and also the number of lines you want to use for column headers.

Delimiters

Most text files use a unique character called a *delimiter* or column separator to mark the separation between items of data. For example, data in a comma-separated value (CSV) file is, of course, separated by commas. Data in some other file might be separated by tab or space characters.

When the Import Wizard imports from a text file or the clipboard, it makes its best guess as to which character was intended as the delimiter and displays the data accordingly. If this is not correct, you will need to set the correct delimiter from the choices shown under **Select Column Separator(s)** in the upper left of the dialog box. When you do this, the Import Wizard immediately reformats the data, displaying new values for the data shown in the preview pane.

Header Format

When reading in data from a text file or the clipboard, the Wizard looks for any lines at the top that have no numeric characters, and assigns these lines to the variable `textdata`. MATLAB counts these lines and displays the count in the value field of **Number of text header lines** in the upper right of the Import Wizard window. You can adjust this count if it does not accurately represent the header format within the file.

Note The **Number of text header lines** selector applies only to column headers. It has no effect on row headers.

MATLAB creates a row vector from the bottommost of these lines and assigns it to the variable `colheaders`.

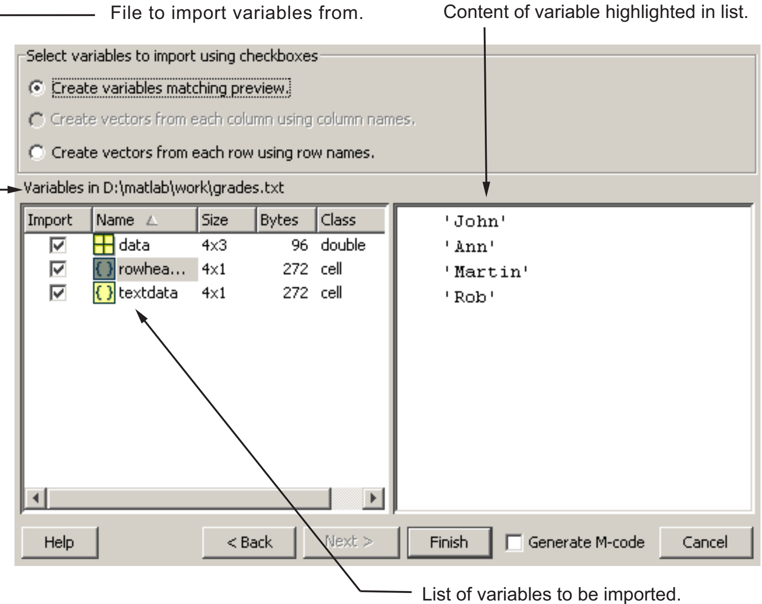
Generate M-Code Checkbox

The **Generate M-code** checkbox at the bottom of the Import Wizard dialog box applies to both text and binary data, and thus is described in "Automated M-Code Generation" on page 6-18.

To continue, click **Next** at the bottom of the dialog box.

Determining Assignment to Variables

At this point, the Import Wizard displays the dialog box shown below. This dialog displays data for both text and binary files.



The left pane of the dialog box displays a list of the variables MATLAB created for your data. For text files, MATLAB derives the variable names as described in step 2: Preview Contents of the File. For binary files, the variable names are taken directly from the file.

Click any variable name and MATLAB displays the contents of that variable in the pane to the right. MATLAB highlights the name of the variable that is currently displayed in the right pane.

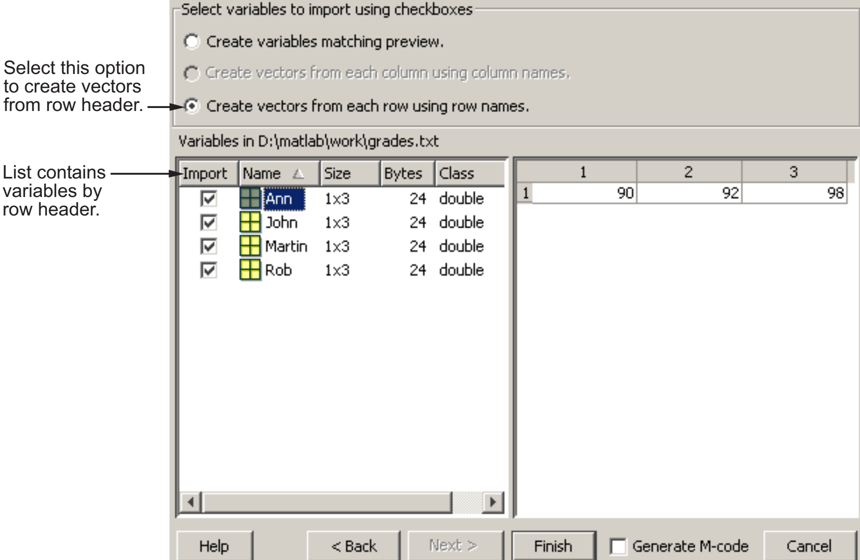
Structuring the Output Data

The top portion of this dialog box offers three options for organizing the file's data:

- **Create variables matching preview**
- **Create vectors from each column using column names**
- **Create vectors from each row using row names**

Note For data imported from a binary file, only the top option is active. Variable names and assignment are taken directly from the imported file. For text data, you can use any of the three options, however, the bottom two are active only if the file or clipboard contains row or column headers.

While importing from the example text file `grades.txt`, select the third option to create vectors from row names. Observe that the display replaces the default variable assignments with new variables derived from the row headers. Click any of these variable names, and the Wizard displays the contents of the corresponding row vector.



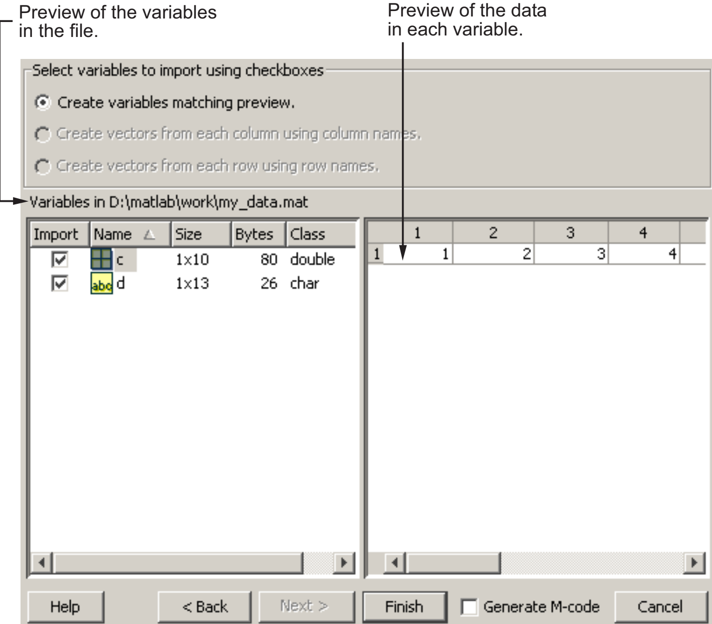
Selecting Which Variables to Write to the Workspace

The checkboxes to the left of each variable name enable you to include or exclude individual variables from those that will be written to the workspace. By default, all variables are selected. Select the checkbox of any variable you do not want written to the workspace. The check mark is removed from any variables that you deselect.

Example of Selecting Variables to Load. Use the Import Wizard to import this sample binary MAT-file, `my_data.mat`,

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

D =
     a test string
```

The Import Wizard displays two variables, as listed in the preview pane. To select a variable to import, select the check box next to its name. All variables are preselected by default.



Automated M-Code Generation

To perform additional imports from this or a similar type of file, you can automate this process by creating a MATLAB function that performs all of the steps you just went through. To have the Import Wizard write this function

for you, select the **Generate M-code** checkbox in the lower right corner of the Wizard dialog.

Once you click **Finish** to complete the import, MATLAB opens an Editor window displaying the generated M-file function. The function is called `importfile.m`. If this name is already taken, then MATLAB names the file `importfileN.m`, where `N` is a number that is one greater than the highest existing `importfile.m` file.

The generated function has the following input and output arguments:

- Input: `fileToRead1`—Name of the file to import from. This argument exists only when importing from a file.
- Output: `newData1`—Structure to assign all imported data to. This argument exists only if you have specified an output argument with the call to `uiimport` when starting the Import Wizard. Otherwise, variables retain the same naming as assigned within the Wizard.

The `newData1` output is a structure that has one field for each output of the import operation.

The workspace variables created by this generated M-code are the same as those created by running the Import Wizard. For example, if you elect to format the output in column vectors when running the Import Wizard, the generated M-file does the same. However, unlike the Import Wizard, you cannot mark any variables to be excluded from the output.

Make any necessary modifications to the generated M-file function in the Editor window. To save the M-file, select **Save** from the **File** menu at the top.

Caution You must save the file yourself; MATLAB does not automatically save it for you.

Example of M-Code Generation

The M-file shown below was generated by MATLAB during an import of the file `grades.txt`, shown earlier in this section. During the import that created this file, the option to **Create vectors from each row using row names**

was selected, thus generating four row vectors for output: Ann, John, Martin, and Rob. Also, the row vector for John was deselected by clearing the checkbox next to that name in the Wizard.

```
function importfile(fileToRead1)
%IMPORTFILE(FILETOREAD1)
%  Imports data from the specified file
%  FILETOREAD1:  file to read

%  Auto-generated by MATLAB on 23-May-2006 14:16:32

% Import the file
newData1 = importdata(fileToRead1);

% Break the data up into a new structure with one field per row.
rowheaders = genvarname(newData1.rowheaders);
for i = 1:length(rowheaders)
    dataByRow1.(rowheaders{i}) = newData1.data(i, :);
end

% Create new variables in the base workspace from those fields.
vars = fieldnames(dataByRow1);
for i = 1:length(vars)
    assignin('base', vars{i}, dataByRow1.(vars{i}));
end
```

If you run the function, you find that the workspace now holds the four row vectors Ann, John, Martin, and Rob, instead of the default variables created by the Import Wizard (data, textdata, and rowheaders). Also, note that the vector for John is written to the workspace along with the others, even though this one variable had been deselected from the Import Wizard interface.

```
importfile grades.txt

whos
  Name          Size            Bytes  Class     Attributes

  Ann           1x3                24  double
```

```
John        1x3                    24  double
Martin      1x3                    24  double
Rob         1x3                    24  double
```

Writing Data to the Workspace

To complete the import operation, click **Finish** to bring the data into the MATLAB workspace. This button also dismisses the Import Wizard.

Variables written to the workspace are in one of the following formats. The first three apply only to data read from text files or the clipboard, the fourth applies only to binary files, and the last applies to both:

Variable Name	Output
`data`, `textdata`, `rowheaders`, `colheaders`	Separate matrices for numeric, text, and header data.
Variables named after row or column headers	One vector for each row or column.
Single variable named after the filename, or `A_pastespecial`	One matrix for all data named after the filename
Variable names taken from binary file	Data assigned to each variable stored in a binary file.
Output variable assigned during call to `uiimport`	A single structure having fields that match one of the formats described above.

Examples

Here are a few examples of how to use the Import Wizard.

Example 1—Text Data. Start by creating the text file `grades.txt` using the MATLAB editor. The file contains the following:

```
John      85     90     95
Ann       90     92     98
Martin   100     95     97
Rob       77     86     93
```

Import from text file `grades.txt`, using default variables to store the data:

```
uiimport grades.txt
whos
  Name               Size              Bytes  Class     Attributes

  data               4x3                  96  double
  rowheaders         4x1                 272  cell
  textdata           4x1                 272  cell
```

Example 2—Partial Text File with Row Vectors. Import from the same file as in the above example, but this time select **Create vectors from each row using row names**. Also, clear the checkbox next to variable `John` so that this one vector does not get written to the workspace:

```
whos
  Name          Size              Bytes  Class     Attributes

  Ann           1x3                  24  double
  Martin        1x3                  24  double
  Rob           1x3                  24  double
```

Example 3—Binary Data Assigned to a Structure. Save numeric and text data in binary format in file `importtest.mat` and use the Import Wizard to import the binary file into the workspace.

```
C = [1 2 3 4 5;6 7 8 9 10];
D = 'a test string';
save importtest C D

clear
s = uiimport('importtest.mat')
s =
    C: [2x5 double]
    D: 'a test string'
```

Accessing Files with Memory-Mapping

In this section...

"Overview of Memory-Mapping in MATLAB" on page 6-23

"The memmapfile Class" on page 6-27

"Constructing a memmapfile Object" on page 6-29

"Reading a Mapped File" on page 6-43

"Writing to a Mapped File" on page 6-48

"Methods of the memmapfile Class" on page 6-56

"Deleting a Memory Map" on page 6-58

"Memory-Mapping Demo" on page 6-58

Overview of Memory-Mapping in MATLAB

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. This makes file reads and writes faster in comparison with using functions such as `fread` and `fwrite`.

Another advantage of using memory-mapping in MATLAB is that it enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file.

This section describes the benefits and limitations of memory-mapping in MATLAB. The last part of this section gives details on which types of applications derive the greatest advantage from using memory-mapping:

- "Benefits of Memory-Mapping" on page 6-24
- "Limitations of Memory-Mapping in MATLAB" on page 6-25
- "When to Use Memory-Mapping" on page 6-26

Benefits of Memory-Mapping

The principal benefits of memory-mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding.

Faster File Access. Accessing files via memory map is faster than using I/O functions such as `fread` and `fwrite`. Data is read and written using the virtual memory capabilities that are built in to the operating system rather than having to allocate, copy into, and then deallocate data buffers owned by the process.

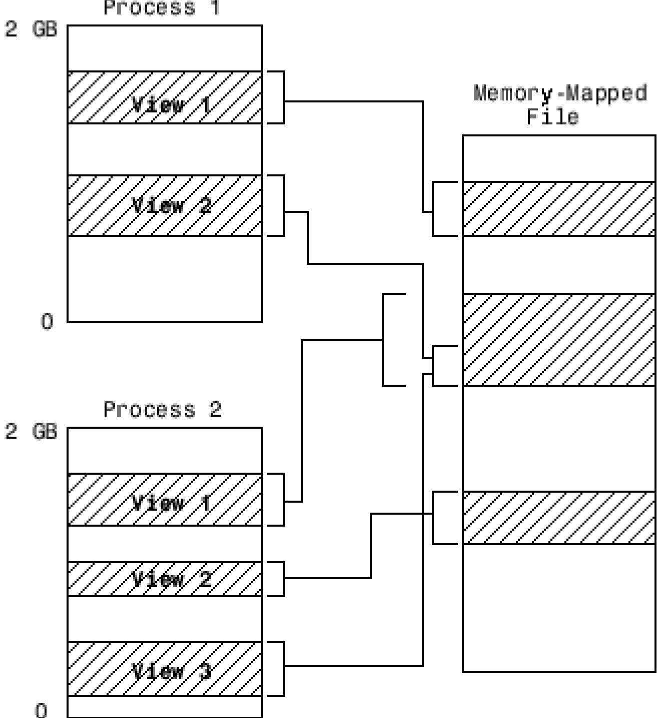
MATLAB does not access data from the disk when the map is first constructed. It only reads or writes the file on disk when a specified part of the memory map is accessed, and then it only reads that specific part. This provides faster random access to the mapped data.

Efficiency. Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space. Initially, MATLAB only allocates address space for the array; it does not actually read data from the file until you access the mapped region. As a result, memory-mapped files provide a mechanism by which applications can access data segments in an extremely large file without having to read the entire file into memory first.

Efficient Coding Style. Memory-mapping eliminates the need for explicit calls to the `fread` and `fwrite` functions. In MATLAB, if `x` is a memory-mapped variable, and `y` is the data to be written to a file, then writing to the file is as simple as

```
x.Data = y;
```

Sharing Memory Between Applications. Memory-mapped files also provide a mechanism for sharing data between applications, as shown in the figure below. This is achieved by having each application map sections of the same file. This feature can be used to transfer large data sets between MATLAB and other applications.



Also, within a single application, you can map the same segment of a file more than once.

Limitations of Memory-Mapping in MATLAB

MATLAB restricts the size of a memory map to 2 gigabytes, and on some platforms, requires that you set up your memory-mapping so that all data access is aligned properly. See the following section, "Maximum Size of a Memory Map", for more information.

Maximum Size of a Memory Map. Due to limits set by the operating system, the maximum amount of data you can map with a single instance of a memory map is 2^{31} - 1 (or 2 GB). If you need to map more than 2 GB, you can either create separate maps for different regions of the file, or you can move the 2 GB window of one map to different locations in the file.

The 2 GB limit also applies to 64-bit platforms. However, because 64-bit platforms have a much larger address space, they can support having many more map instances in memory at any given time.

Aligned Access on Sol64. The Sol64 platform only supports aligned data access. This means that numeric values of type `double` that are to be read from a memory-mapped file must start at some multiple of 8 bytes from the start of the file. (Note that this is from the start of the *file*, and not the start of the mapped region.) Furthermore, numeric values of type `single` and also 32-bit integers must start at multiples of 4 bytes, and 16-bit integers at 2-byte multiples.

If you attempt to map a file on Sol64 that does not take into account these alignment considerations, MATLAB generates an error.

Byte Ordering

Memory-mapping works only with data that has the same byte ordering scheme as the native byte ordering of your operating system. For example, because both Linux and Windows use little-endian byte ordering, data created on a Linux system can be read on Windows. You can use the `computer` function to determine the native byte ordering of your current system.

When to Use Memory-Mapping

Just how much advantage you get from mapping a file to memory depends mostly on the size and format of the file, the way in which data in the file is used, and the computer platform you are using.

When Memory-Mapping Is Most Useful. Memory-mapping works best with binary files, and in the following scenarios:

- For large files that you want to access randomly one or more times

- For small files that you want to read into memory once and access frequently
- For data that you want to share between applications
- When you want to work with data in a file as if it were a MATLAB array

When the Advantage Is Less Significant. The following types of files do not fully utilize the benefits of memory-mapping:

- Formatted binary files like HDF or TIFF that require customized readers are not good for memory-mapping. For one thing, describing the data contained in these files can be a very complex task. Also, you cannot access data directly from the mapped segment, but must instead create arrays to hold the data.
- Text or ASCII files require that you convert the text in the mapped region to an appropriate type for the data to be meaningful. This takes up additional address space.
- Files that are larger than several hundred megabytes in size consume a significant amount of the virtual address space needed by MATLAB to process your program. Mapping files of this size may result in MATLAB reporting out-of-memory errors more often. This is more likely if MATLAB has been running for some time, or if the memory used by MATLAB becomes fragmented.

The memmapfile Class

MATLAB implements memory-mapping using an object-oriented class called `memmapfile`. The `memmapfile` class has the properties and methods you need to map to a file, read and write the file via the map, and remove the map from memory when you are done.

Properties of the memmapfile Class

There are six properties defined for the `memmapfile` class. These are shown in the table below. These properties control which file is being mapped, where in the file the mapping is to begin and end, how the contents of the file are to be formatted, and whether or not the file is writable. One property of the file contains the file data itself.

Property	Description	Data Type	Default
`Data`	Contains the data read from the file or to be written to the file. (See “Reading a Mapped File” on page 6-43 and “Writing to a Mapped File” on page 6-48)	Any of the numeric types	None
`Filename`	Path and name of the file to map into memory. (See “Selecting the File to Map” on page 6-32)	`char` array	None
`Format`	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which to access the data. (See “Identifying the Contents of the Mapped Region” on page 6-34)	`char` array or N-by-3 cell array	`uint8`
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file. Must be a nonnegative integer value. (See “Setting the Start of the Mapped Region” on page 6-34)	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file. Must be a positive integer value or `Inf`. (See “Repeating a Format Scheme” on page 6-41)	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region. Must be logical 1 or logical 0. (See “Setting the Type of Access” on page 6-42)	`logical`	`false`

You can set the values for any property except for `Data` at the time you call the `memmapfile` constructor, or at any time after that while the map is still valid. Any properties that are not explicitly set when you construct the object are given their default values as shown in the table above. For information on calling the constructor, see “Constructing a memmapfile Object” on page 6-29.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. For example, to set a new `Offset` value for memory map object `m`, type

```
m.Offset = 2048;
```

Note Property names are not case sensitive. For example, MATLAB considers `m.offset` to be the same as `m.Offset`.

To display the value of all properties of a `memmapfile` object, simply type the object name. For a `memmapfile` object `m`, typing the variable name `m` displays the following. Note that this example requires the file `records.dat` which you will create at the beginning of the next section.

```
m =
    Filename: 'records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint32'
      Repeat: Inf
        Data: 4778x1 uint32 array
```

To display the value of any individual property, for example the `Writable` property of object `m`, type

```
m.Writable
ans =
   true
```

Constructing a memmapfile Object

The first step in mapping to any file is to construct an instance of the `memmapfile` class using the class constructor function. You can have MATLAB assign default values to each of the new object's properties, or you can specify property values yourself in the call to the `memmapfile` constructor.

For information on how to set these values, see the sections that cover

- "Constructing the Object with Default Property Values" on page 6-30
- "Changing Property Values" on page 6-31
- "Selecting the File to Map" on page 6-32
- "Setting the Start of the Mapped Region" on page 6-34
- "Identifying the Contents of the Mapped Region" on page 6-34

- “Mapping of the Example File” on page 6-39
- “Repeating a Format Scheme” on page 6-41
- “Setting the Type of Access” on page 6-42

All the examples in this section use a file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating point numbers. Use the following code to generate this file before going on to the next sections of this documentation.

First, save this function in your current working directory:

```
function gendatafile(filename, count)
dmax32 = double(intmax('uint32'));
rand('state', 0)

fid = fopen(filename,'w');
fwrite(fid, rand(count,1)*dmax32, 'double');
fclose(fid);
```

Now execute the `gendatafile` function to generate the `records.dat` file that is referenced in this section. You can use this function at any time to regenerate the file:

```
gendatafile('records.dat', 5000);
```

Constructing the Object with Default Property Values

The simplest and most general way to call the constructor is with one input argument that specifies the name of the file you want to map. All other properties are optional and are given their default values. Use the syntax shown here:

```
objname = memmapfile(filename)
```

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat')
m =
    Filename: 'd:\matlab\mfiles\records.dat'
```

```
    Writable: false
      Offset: 0
      Format: 'uint8'
      Repeat: Inf
        Data: 40000x1 uint8 array
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers, and gives the caller read-only access to its contents.

Changing Property Values

You can make the memory map more specific to your needs by including more information when calling the constructor. In addition to the `filename` argument, there are four other parameters that you can pass to the constructor. Each of these parameters represents a property of the object, and each requires an accompanying value to be passed as well:

```
objname = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

For example, to construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names and string parameter values in quotes:

```
m = memmapfile('records.dat', ...
      'Offset', 1024,         ...
      'Format', 'double',     ...
      'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'double'
      Repeat: Inf
```

```
          Data: 4872x1 double array
```

You can also change the value of any property after the object has been constructed. Use the syntax

```
objname.property = newvalue;
```

For example, to set the format to `uint16`, type the following. (Property names, like `Format`, are not case sensitive.)

```
m.format = 'uint16'
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint16'
      Repeat: Inf
        Data: 19488x1 uint16 array
```

Further read and write operations to the region mapped by `m` will now treat the data in the file as a sequence of unsigned 16-bit integers. Whenever you change the value of a `memmapfile` property, MATLAB remaps the file to memory.

Selecting the File to Map

`filename` is the only required argument when you call the `memmapfile` constructor. When you call the `memmapfile` constructor, MATLAB assigns the filename that you specify to the `Filename` property of the new object instance.

Specify the filename as a quoted string, (e.g., `'records.dat'`). It must be first in the argument list and not specified as a parameter-value pair. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), and is not case sensitive.

Note Unlike the other `memmapfile` constructor arguments, you must specify `filename` as a single string, and not as a parameter-value pair.

If the file to be mapped resides somewhere on the MATLAB path, you can use a partial pathname. If the path to the file is not fully specified, MATLAB searches for the file in your current working directory first, and then on the MATLAB path.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

You can change the value of the `Filename` property at any time after constructing the `memmapfile` object. You might want to do this if

- You want to use the same `memmapfile` object on more than one file.
- You save your `memmapfile` object to a MAT-file, and then later load it back into MATLAB in an environment where the mapped file has been moved to a different location. This requires that you modify the path segment of the `Filename` string to represent the new location.

For example, save `memmapfile` object `m` to file `mymap.mat`:

```
disp(m.Filename)
    d:\matlab\mfiles\records.dat

save mymat m
```

Now move the file to another location, load the object back into MATLAB, and update the path in the `Filename` property:

```
load mymat m
m.Filename = 'f:\testfiles\oct1\records.dat'
```

Note You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Setting the Start of the Mapped Region

By default, MATLAB begins a memory map at the start of the file. To begin the mapped region at some point beyond the start of the file, specify an `Offset` parameter in the call to the `memmapfile` constructor:

```
objname = memmapfile(filename, 'Offset', bytecount)
```

The `bytecount` value is the number of bytes from the beginning of the file to the point in the file where you want the memory map to start (a zero-based offset). To map the file `records.dat` from a point 1024 bytes from the start and extending to the end of the file, type

```
m = memmapfile('records.dat', 'Offset', 1024);
```

You can change the starting position of an existing memory map by setting the `Offset` property for the associated object to a new value. The following command sets the offset of `memmapfile` object `m` to be 2,048 bytes from the start of the mapped file:

```
m.Offset = 2048;
```

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` offset on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See "Aligned Access on Sol64" on page 6-26).

Identifying the Contents of the Mapped Region

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. To have the data interpreted otherwise as it is read or written to in the mapped file, specify a `Format` parameter and value in your call to the constructor:

```
objname = memmapfile(filename, 'Format', formatspec)
```

The `formatspec` argument can either be a character string that identifies a single data type used throughout the mapped region, or a cell array that specifies more than one data type.

For example, say that you map a file that is 12K bytes in length. Data read from this file could be treated as a sequence of 6,000 16-bit (2-byte) integers, or as 1,500 8-byte double-precision floating-point numbers, to name just a couple of possibilities. Or you could read this data in as a combination of different types: for example, as 4,000 8-bit (1-byte) integers followed by 1,000 64-bit (8-byte) integers. You determine how MATLAB will interpret the mapped data by setting the `Format` property of the `memmapfile` object when you call its constructor.

Note MATLAB arrays are stored on disk in column-major order. (The sequence of array elements is column 1, row 1; column 1, row 2; column 1, last row; column 2, row 1, and so on.) You might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` format on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See "Aligned Access on Sol64" on page 6-26).

Data types supported for the `Format` property are shown at the end of this section. See "Supported Data Types for the Format Property" on page 6-40.

For more information on format options see

- "Mapping a Single Data Type" on page 6-35
- "Formatting the Mapped Data to an Array" on page 6-36
- "Mapping Multiple Data Types and Arrays" on page 6-38

Mapping a Single Data Type. If the file region being mapped contains data of only one type, specify the `Format` value as a character string identifying that type:

```
objname = memmapfile(filename, 'Format', datatype)
```

The following command constructs a memmapfile object for the entire file records.dat, and sets the Format property for that object to uint64. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64')
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint64'
      Repeat: Inf
        Data: 5000x1 uint64 array
```

You can change the value of the Format property at any time after the memmapfile object is constructed. Use the object.property syntax shown here in assigning the new value:

```
m.Format = 'int32';
```

Further read and write operations to the region mapped by m will now treat the data in the file as a sequence of signed 32-bit integers.

Property names, like Format, are not case sensitive.

Formatting the Mapped Data to an Array. You can also specify an array shape to be applied to the data read or written to the mapped file, and a field name to be used in referencing this array. Use a cell array to hold these values either when calling the memmapfile constructor or when modifying m.Format after the object has been constructed. The cell array contains three elements: the data type to be applied to the mapped region, the dimensions of the array shape that is applied to the region, and a field name to use in referencing the data:

```
objname = memmapfile(filename, ...
            'Format', {datatype, dimensions, varname})
```

The command below constructs a memmapfile object for a region of records.dat such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name x:

```
m = memmapfile('records.dat',   ...
      'Offset', 1024,           ...
      'Format', {'uint32' [4 10 18] 'x'})
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 1024
      Format: {'uint32' [4 10 18] 'x'}
      Repeat: Inf
        Data: 13x1 struct array with fields:
                x

A = m.Data(1).x;

whos A
  Name      Size                           Bytes  Class

  A        4x10x18                         2880   uint32 array

Grand total is 720 elements using 2880 bytes
```

You can change the data type, array shape, or field name that MATLAB applies to the mapped region at any time by setting a new value for the `Format` property of the object:

```
m.Format = {'uint64' [30 4 10] 'x'};
A = m.Data(1).x;

whos A
  Name      Size                           Bytes  Class

  A        30x4x10                          9600  uint64 array

Grand total is 1200 elements using 9600 bytes
```

Mapping Multiple Data Types and Arrays. If the region being mapped is composed of segments of varying data types or array shapes, you can specify an individual format for each segment using an N-by-3 cell array, where N is the number of segments. The cells of each cell array row identify the data type for that segment, the array dimensions to map the data to, and a field name by which to reference that segment:

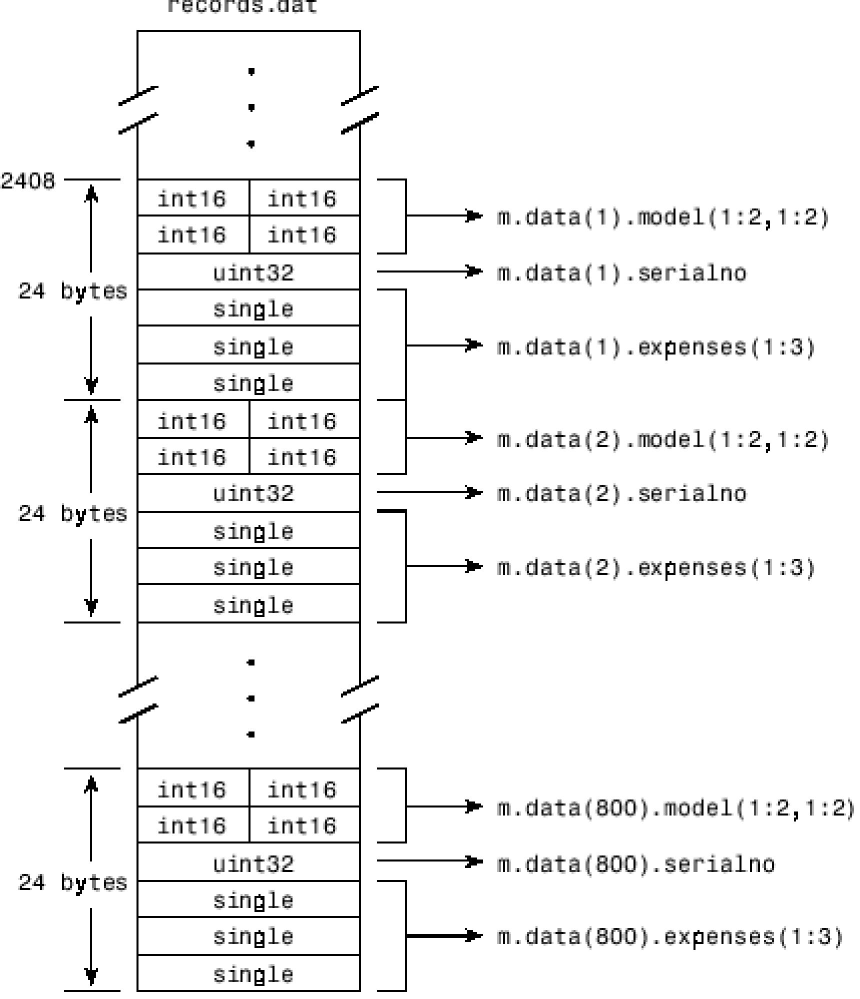
```
objname = memmapfile(filename,                              ...
              'Format', {                                   ...
                 datatype1, dimensions1, fieldname1;        ...
                 datatype2, dimensions2, fieldname2;        ...
                     :           :            :             ...
                 datatypeN, dimensionsN, fieldnameN})
```

The following command maps a 24 kilobyte file containing data of three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

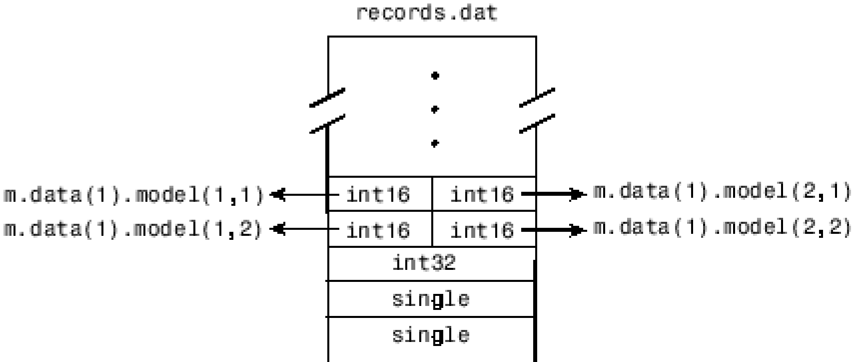
Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',        ...
      'Offset', 2048,                ...
      'Format', {                    ...
         'int16'  [2 2] 'model';     ...
         'uint32' [1 1] 'serialno';  ...
         'single' [1 3] 'expenses'});
```

Mapping of the Example File



The figure below shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.



If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Supported Data Types for the Format Property. Any of the following data types can be used when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
`'int8'`	Signed 8-bit integers
`'int16'`	Signed 16-bit integers
`'int32'`	Signed 32-bit integers
`'int64'`	Signed 64-bit integers
`'uint8'`	Unsigned 8-bit integers
`'uint16'`	Unsigned 16-bit integers
`'uint32'`	Unsigned 32-bit integers
`'uint64'`	Unsigned 64-bit integers
`'single'`	32-bit floating-point
`'double'`	64-bit floating-point

Repeating a Format Scheme

Once you have set a `Format` value for the `memmapfile` object, you can have MATLAB apply that format to the file data multiple times by specifying a `Repeat` value when you call the `memmapfile` constructor:

```
objname = memmapfile(filename,    ...
             'Format', formatspec, ...
             'Repeat', count)
```

The `Repeat` value applies to the whole format specifier, whether that specifier describes just a single data type that repeats, or a more complex format that includes various data types and array shapes. The default `Repeat` value is infinity (`inf`), which means that the full extent of the `Format` specifier repeats as many times as possible within the mapped region.

The next example maps a file region identical to that of the previous example, except the pattern of `int16`, `uint32`, and `single` data types is repeated only three times within the mapped region of the file:

```
m = memmapfile('records.dat',         ...
       'Offset', 2048,                ...
       'Format', {                    ...
          'int16'  [2 2] 'model';      ...
          'uint32' [1 1] 'serialno';   ...
          'single' [1 3] 'expenses'},  ...
       'Repeat', 3);
```

You can change the value of the `Repeat` property at any time. To change the repeat value to 5, type

```
m.Repeat = 5;
```

Property names, like `Repeat`, are not case sensitive.

Keeping the Repeated Format Within the Mapped Region. MATLAB maps only the *full* pattern specified by the `Format` property. If you repeat a format such that it would cause the map to extend beyond the end of the file, then either of two things can happen:

- If you specify a repeat value of `Inf`, then only those repeated segments that fit within the file in their entirety are applied to the map.

- If you specify a repeat value other than `Inf`, and that value would cause the map to extend beyond the end of the file, then MATLAB generates an error.

Considering the last example, if the part of the file from `m.Offset` to the end were 70 bytes (instead of the 72 bytes required to repeat `m.Format` three times) and you used a `Repeat` value of `Inf`, then only two full repetitions of the specified format would have been mapped. The end result would be as if you had constructed the map with this command:

```
m = memmapfile('records.dat',          ...
      'Offset', 2048,                  ...
      'Format', {                      ...
         'int16'  [2 2] 'model';       ...
         'uint32' [1 1] 'serialno';    ...
         'single' [1 3] 'expenses'},   ...
      'Repeat', 2);
```

If `Repeat` were set to 3 and you had only 70 bytes to the end of the file, you would get an error.

Note `memmapfile` does not expand or append to a mapped file. Use standard file I/O functions like `fopen` and `fwrite` to do this.

Setting the Type of Access

You can map a file region to allow either read-only or read and write access to its contents. Pass a `Writable` parameter and value in the `memmapfile` constructor, or set `m.Writable` on an existing object to set the type of access allowed:

```
objname = memmapfile(filename, 'Writable', trueorfalse)
```

The value passed can be either `true` (equal to `logical(1)`) or `false` (equal to `logical(0)`). By default, it is `false`, meaning that the mapped region is read only.

To map a read and write region of the file `records.dat` in memory, type

```
m = memmapfile('records.dat', 'Writable', true);
```

Note To successfully modify the file you are mapping to, you must have write permission for that file. If you do not have write permission, you can still set the `Writable` property to `true`, but attempting to write to the file generates an error.

You can change the value of the `Writable` property at any time. To make the memory map to `records.dat` read only, type

```
m.Writable = false;
```

Property names, like `Writable`, are not case sensitive.

Reading a Mapped File

The most commonly used property of the `memmapfile` class is the `Data` property. It is through this property of the memory-map object that MATLAB provides all read and write access to the contents of the mapped file.

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

Once you have mapped a file to memory, you can read the contents of that file using the same MATLAB statements used to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map object, the contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read the desired data from the file.

This section covers the following topics:

- “Improving Performance” on page 6-44
- “Example 1 — Reading a Single Data Type” on page 6-44
- “Example 2 — Formatting File Data as a Matrix” on page 6-45
- “Example 3 — Reading Multiple Data Types” on page 6-46

- “Example 4 — Modifying Map Parameters” on page 6-47

Improving Performance

MATLAB accesses data in structures more efficiently than it does data contained in objects. The main reason is that structures do not require the extra overhead of a `subsref` routine. Instead of reading directly from the `memmapfile` object, as shown here

```
for k = 1 : N
   y(k) = m.Data(k);
end
```

you will get better performance when you assign the `Data` field to a variable and then read or write the mapped file through this variable, as shown in this second example:

```
dataRef = m.Data;
for k = 1 : N
   y(k) = dataRef(k);
end
```

Example 1 — Reading a Single Data Type

This example maps a file of 100 double-precision floating-point numbers to memory. The map begins 1024 bytes from the start of the file, and ends 800 bytes (8 bytes per `double` times a `Repeat` value of 100) from that point.

If you haven’t done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under “Constructing a memmapfile Object” on page 6-29:

```
gendatafile('records.dat', 5000);
```

Now, construct the `memmapfile` object `m`, and show the format of its `Data` property:

```
m = memmapfile('records.dat', 'Format', 'double', ...
      'Offset', 1024, 'Repeat', 100);
```

```
d = m.Data;

whos d
  Name      Size                      Bytes  Class

  d       100x1                         800  double array

Grand total is 100 elements using 800 bytes
```

Read a selected set of numbers from the file by indexing into the single-precision array `m.Data`:

```
d(15:20)
ans =
  1.0e+009 *
    3.6045
    2.7006
    0.5745
    0.8896
    2.6079
    2.7053
```

Example 2 — Formatting File Data as a Matrix

This example is similar to the last, except that the constructor of the `memmapfile` object now specifies an array shape of 4-by-6 to be applied to the data as it is read from the mapped file. MATLAB maps the file contents into a structure array rather than a numeric array, as in the previous example:

```
m = memmapfile('records.dat',           ...
      'Format', {'double', [4 6], 'x'},  ...
      'Offset', 1024, 'Repeat', 100);

d = m.Data;

whos d
  Name      Size                      Bytes  Class

  d       100x1                       25264  struct array

Grand total is 2500 elements using 25264 bytes
```

When you read an element of the structure array, MATLAB presents the data in the form of a 4-by-6 array:

```
d(5).x
ans =
  1.0e+009 *
    3.1564    0.6684    2.1056    1.9357    1.2773    4.2219
    2.9520    0.8208    3.5044    1.7705    0.2112    2.3737
    1.4865    1.8144    1.9790    3.8724    2.9772    1.7183
    0.7131    3.6764    1.9643    0.0240    2.7922    0.8538
```

To index into the structure array field, use

```
d(5).x(3,2:6)
ans =
  1.0e+009 *
    1.8144    1.9790    3.8724    2.9772    1.7183
```

Example 3 — Reading Multiple Data Types

This example maps a file containing more than one data type. The different data types contained in the file are mapped as fields of the returned structure array `m.Data`.

The `Format` parameter passed in the constructor specifies that the first 80 bytes of the file are to be treated as a 5-by-8 matrix of `uint16`, and the 160 bytes after that as a 4-by-5 matrix of `double`. This pattern repeats until the end of the file is reached. The example shows different ways of reading the `Data` property of the object.

Start by calling the `memmapfile` constructor to create a memory map object, `m`:

```
m = memmapfile('records.dat',  ...
      'Format', {              ...
         'uint16' [5 8] 'x';   ...
         'double' [4 5] 'y' });
```

If you examine the `Data` property, MATLAB shows a 166-element structure array with two fields, one for each format specifier in the constructor:

```
d = m.Data
```

```
ans =
166x1 struct array with fields:
    x
    y
```

Examine one structure in the array to show the format of each field:

```
d(3)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

Now read the `x` and `y` fields of that structure from the file. MATLAB formats the first block of data as a 5-by-8 matrix of `uint16`, as specified in the `Format` property, and the second block as a 4-by-5 matrix of `double`:

```
d(3).x
ans =
  34432  47500  19145  16868  38165  47956  35550  16853
  60654  51944  16874  47166  35397  58072  16850  56576
  51075  16876  12471  34369   8341  16853  44509  57652
  16863  16453   6666  11480  16869  58695  36217   5932
  57883  15551  41755  16874  37774  31693  54813  16865

d(3).y
ans =
  1.0e+009 *
    3.1229    1.5909    2.9831    2.2445    1.1659
    1.3284    3.0182    2.6685    3.7802    1.0837
    3.6013    2.3475    3.4137    0.7428    3.7613
    2.4399    1.9107    4.1096    4.2080    3.1667
```

Example 4 — Modifying Map Parameters

This example plots the Fourier transform output of data read from a file via a memory map. It then modifies several parameters of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a memory-mapped object, mapping 1,000 elements of type `double` starting at the 1025th byte:

```
m = memmapfile('mybinary.bin', 'Offset', 1024,  ...
      'Format', 'double', 'Repeat', 1000);
```

Get data associated with the map and plot the FFT of the first 1000 values of the map. This is when the map is actually created, because no data has been referenced until this point:

```
plot(abs(fft(m.Data(1:1000))));
```

Get information about the memory map:

```
mapStruct = get(m)

mapStruct =
    Filename: 'd:\matlab\mfiles\mybinary.bin'
    Writable: 0
      Offset: 1024
      Format: 'double'
      Repeat: 1000
        Data: [1000x1 double]
```

Change the map, but continue using the same file:

```
m.Offset = 4096;
m.Format = 'single';
m.Repeat = 800;
```

Read from a different area of the file, and plot a histogram of the data. This maps a new region and unmaps the previous region:

```
hist(m.Data)
```

Writing to a Mapped File

Writing to a mapped file is done with standard MATLAB subscripted assignment commands. To write to a particular location in the file mapped to `memmapfile` object `m`, assign the value to the `m.Data` structure array index and field that map to that location.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under "Constructing a memmapfile Object" on page 6-29:

```
gendatafile('records.dat', 5000);
```

Now call the `memmapfile` constructor to create the object:

```
m = memmapfile('records.dat',  ...
      'Format', {              ...
         'uint16' [5 8] 'x';   ...
         'double' [4 5] 'y' });
```

If you are going to modify the mapped file, be sure that you have write permission, and that you set the `Writable` property of the `memmapfile` object to `true` (logical 1):

```
m.Writable = true;
```

(You do not have to set `Writable` as a separate command, as done here. You can include a `Writable` parameter-value argument in the call to the `memmapfile` constructor.)

Read from the 5-by-8 matrix `x` at `m.Data(2)`:

```
d = m.Data;

d(2).x
ans =
  35330   4902  31861  16877  23791  61500  52748  16841
  51314  58795  16860  43523   8957   5182  16864  60110
  18415  16871  59373  61001  52007  16875  26374  28570
  16783   4356  52847  53977  16858  38427  16067  33318
  65372  48883  53612  16861  18882  39824  61529  16869
```

Update all values in that matrix using a standard MATLAB assignment statement:

```
d(2).x = d(2).x * 1.5;
```

Verify the results:

```
d(2).x
ans =
  52995   7353  47792  25316  35687  65535  65535  25262
  65535  65535  25290  65285  13436   7773  25296  65535
  27623  25307  65535  65535  65535  25313  39561  42855
  25175   6534  65535  65535  25287  57641  24101  49977
  65535  65535  65535  25292  28323  59736  65535  25304
```

This section covers the following topics:

- "Dimensions of the Data Field" on page 6-50
- "Writing Matrices to a Mapped File" on page 6-51
- "Selecting Appropriate Data Types" on page 6-54
- "Working with Copies of the Mapped Data" on page 6-54
- "Invalid Syntax for Writing to Mapped Memory" on page 6-55

Dimensions of the Data Field

The dimensions of a `memmapfile` object's `Data` field are set at the time you construct the object and cannot be changed. This differs from other MATLAB arrays that have dimensions you can modify using subscripted assignment.

For example, you can add a new column to the field of a MATLAB structure:

```
A.s = ones(4,5);

A.s(:,6) = [1 2 3 4];         % Add new column to A.s
size(A.s)
ans =
     4     6
```

But not to a similar field of a structure that represents data mapped from a file. The following assignment to `m.Data(60).y` does not expand the size of `y`, but instead generates an error:

```
m.Data(60)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

```
m.Data(60).y(:,6) = [1 2 3 4];        % Generates an error.
```

Thus, if you map an entire file and then append to that file after constructing the map, the appended data is not included in the mapped region. If you need to modify the dimensions of data that you have mapped to a `memmapfile` object, you must either modify the `Format` or `Repeat` properties for the object, or reconstruct the object.

Examples. Two examples of statements that attempt to modify the dimensions of a mapped `Data` field are shown here. These statements result in an error.

The first example attempts to diminish the size of the array by removing a row from the mapped array `m.Data`.

```
m.Data(5) = [];
```

The second example attempts to expand the size of a 50-row mapped array `x` by adding another row to it:

```
m.Data(2).x(1:51,31) = 1:51;
```

Writing Matrices to a Mapped File

The syntax to use when writing to mapped memory can depend on what format was used when you mapped memory to the file.

When Memory Is Mapped in Nonstructure Format. When you map a file as a sequence of a single data type (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as a sequence of elements of the same data type, making `m.Data` an array of a nonstructure type.
- The class of `X` is the same as the class of `m.Data`.
- The number of elements in `X` equals the number of elements in `m.Data`.

This example maps a file as a sequence of 16-bit unsigned integers, and then uses the syntax shown above to write a matrix to the file. Map only a small part of the file, using a uint16 format for this segment:

```
m = memmapfile('records.dat', 'Writable', true', ...
    'Offset', 2000, 'Format', 'uint16', 'Repeat', 15);
```

Create a matrix X of the same size and write it to the mapped part of the file:

```
X = uint16(5:5:75);    % Sequence of 5 to 75, counting by fives.
m.data = X;
```

Verify that new values were written to the file:

```
m.offset = 1980;   m.repeat = 35;
reshape(m.data,5,7)'
ans =
  29158  16841  32915  37696    421      % <== At offset 1980
  16868  51434  17455  30645  16871
      5     10     15     20     25      % <== At offset 2000
     30     35     40     45     50
     55     60     65     70     75
  16872  50155  51100  26469  16873
  56776   6257  28746  16877  34374
```

When Memory Is Mapped in Scalar Structure Format. When you map a file as a sequence of a single data type (e.g., a sequence of uint16), you can use the following syntax to write matrix X to the file:

```
m.Data.f = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as containing multiple data types that do not repeat, making m.Data a scalar structure.
- The class of X is the same as the class of m.Data.f.
- The number of elements in X equals that of m.Data.f.

This example maps a file as a 300-by-8 matrix of type uint16 followed by a 200-by-5 matrix of type double, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',     ...
      'Format', {                 ...
         'uint16' [300 8] 'x';    ...
         'double' [200 5] 'y' },  ...
      'Repeat', 1, 'Writable', true);

m.Data.x = ones(300, 8, 'uint16');
```

When Memory Is Mapped in Nonscalar Structure Format. When you map a file as a repeating sequence of multiple data types, you can use the following syntax to write matrix `X` to the file, providing that `k` is a scalar index:

```
m.Data(k).field = X;
```

To do this, the following conditions must be true:

- The file is mapped as containing multiple data types that can repeat, making `m.Data` a nonscalar structure.
- `k` is a scalar index.
- The class of `X` is the same as the class of `m.Data(k).field`.
- The number of elements in `X` equals that of `m.Data(k).field`.

This example maps a file as a matrix of type `uint16` followed by a matrix of type `double` that repeat 20 times, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',    ...
      'Format', {                ...
         'uint16' [25 8] 'x';    ...
         'double' [15 5] 'y' },  ...
      'Repeat', 20, 'Writable', true);

d = m.Data;

d(12).x = ones(25,8,'uint16');
```

You can write to specific elements of field `x` as shown here:

```
d(12).x(3:5,1:end) = uint16(500);
d(12).x(3:5,1:end)
```

```
ans =
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
```

Selecting Appropriate Data Types

All of the usual MATLAB indexing and data type rules apply when assigning values to data via a memory map. The data type that you assign to must be big enough to hold the value being assigned. For example,

```
m = memmapfile('records.dat', 'Format', 'uint8', ...
      'Writable', true);

d = m.Data;
d(5) = 300;
```

saturates the `x` variable because `x` is defined as an 8-bit integer:

```
d(5)
ans =
   255
```

Working with Copies of the Mapped Data

In the following code, the data in variable `block2` is a *copy* of the file data mapped by `m.Data(2)`. Because it is a copy, modifying array data in `block2` does not modify the data contained in the file:

First, destroy the `memmapfile` object and restore the test file `records.dat`, since it has been modified by running the previous examples:

```
clear m
gendatafile('records.dat',50000);
```

Map the file as a series of `uint16` and `double` matrices and make a copy of `m.Data(2)` in `block2`:

```
m = memmapfile('records.dat',   ...
      'Format', {               ...
         'uint16' [5 8] 'x';    ...
         'double' [4 5] 'y' });
```

```
d = m.Data;
```

Write all zeros to the copy:

```
d(2).x(1:5,1:8) = 0;

d(2).x
ans =
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
```

Verify that the data in the mapped file has not changed even though the copy of `m.Data(2).x` has been written with zeros:

```
m.Data(2).x
ans =
  35330    4902   31861   16877   23791   61500   52748   16841
  51314   58795   16860   43523    8957    5182   16864   60110
  18415   16871   59373   61001   52007   16875   26374   28570
  16783    4356   52847   53977   16858   38427   16067   33318
  65372   48883   53612   16861   18882   39824   61529   16869
```

Invalid Syntax for Writing to Mapped Memory

Although you can expand the dimensions of a typical MATLAB array by assigning outside its current dimensions, this does not apply to the `Data` property of a `memmapfile` object. The following operation is invalid if `m.Data` has only 100 elements:

```
m.Data(120) = x;
```

If you need to expand the size of the mapped data region, first extend the map by updating the `Format` or `Repeat` property of the `memmapfile` object to reflect the new structure of the data in the mapped file.

Methods of the memmapfile Class

You can use the following methods on objects constructed from the `memmapfile` class.

Syntax	Description
`disp`	Displays properties of the object. The display does not include the object's name.
`get(obj)`	Returns the values of all properties of the `memmapfile` object in a structure array.
`get(obj, property)`	Returns the value of the specified property. `property` can be a string or cell array of strings, each containing a property name.

Using the disp Method

Use the `disp` method to display all properties of a `memmapfile` object. The text displayed includes only the property value, and not the object name or the MATLAB response string, `ans =`.

Construct object `m`:

```
m = memmapfile('records.dat',        ...
      'Offset', 2048,                ...
      'Format', {                    ...
         'int16'  [2 2] 'model';     ...
         'uint32' [1 1] 'serialno';  ...
         'single' [1 3] 'expenses'});
```

and display all of its properties:

```
disp(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 2048
      Format: {'int16' [2 2] 'model'
               'uint32' [1 1] 'serialno'
               'single' [1 3] 'expenses'}
      Repeat: Inf
```

```
Data: 16581x1 struct array with fields:
    model
 serialno
 expenses
```

Using the get Method

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Use the `get` method to return the specified property values in a 1-by-3 cell array, `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

You can also choose to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
get(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: 0
      Offset: 2048
      Format: {3x3 cell}
      Repeat: Inf
        Data: [16581x1 struct]
```

Deleting a Memory Map

It is not necessary to explicitly call a destructor method to clear a memmapfile object from memory when you no longer need it. MATLAB calls the destructor for you whenever you do any of the following:

- Reassign another value to the memmapfile object's variable
- Clear the object's variable from memory
- Exit the function scope in which the object was created

The Effect of Shared Data Copies On Performance

When you assign the Data field of the memmapfile object to a variable, MATLAB makes a shared data copy of the mapped data. This is very efficient as no memory actually gets copied. In the following statement, memdat is a shared data copy of the data mapped from the file:

```
memdat = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that shared data with the mapped file before clearing the object itself. If you clear the object first, then the sharing of data between the file and dependent variables is broken, and the data assigned to such variables must be copied into memory before the object is destroyed. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. So, if the statement shown above assigns data to the variable memdat, you should be sure to clear memdat before clearing m when you are finished with the object.

Note Keep in mind that the memmapfile object can be cleared in any of the three ways described under “Deleting a Memory Map” on page 6-58.

Memory-Mapping Demo

In this demonstration, two separate MATLAB processes communicate with each other by writing and reading from a shared file. They share the file by mapping part of their memory space to a common location in the file. A write operation to the memory map belonging to the first process can be read from the map belonging to the second, and vice versa.

One MATLAB process (running `send.m`) writes a message to the file via its memory map. It also writes the length of the message to byte 1 in the file, which serves as a means of notifying the other process that a message is available. The second process (running `answer.m`) monitors byte 1 and, upon seeing it set, displays the received message, puts it into uppercase, and echoes the message back to the sender.

The send Function

This function prompts you to enter a string and then, using memory-mapping, passes the string to another instance of MATLAB that is running the `answer` function.

Copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory. Begin the demonstration by calling `send` with no inputs. Next, start a second MATLAB session on the same machine, and call the `answer` function in this session. To exit, press **Enter**.

```
function send
% Interactively send a message to ANSWER using memmapfile class.

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:send:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Set first byte to zero, indicating a message is not
```

```
        % yet ready.
        m.Data(1) = 0;

        str = input('Enter send string (or RETURN to end): ', 's');

        len = length(str);
        if (len == 0)
            disp('Terminating SEND function.')
            break;
        end

        str = str(1:min(len, 255));  % Message limited to 255 chars.

        % Update the file via the memory map.
        m.Data(2:len+1) = str;
        m.Data(1)=len;

        % Wait until the first byte is set back to zero,
        % indicating that a response is available.
        while (m.Data(1) ~= 0)
            pause(.25);
        end

        % Display the response.
        disp('response from ANSWER is:')
        disp(char(m.Data(2:len+1))')
    end
```

The answer Function

The `answer` function starts a server that, using memory-mapping, watches for a message from `send`. When the message is received, `answer` replaces the message with an uppercase version of it, and sends this new message back to `send`.

To use `answer`, call it with no inputs.

```
function answer
% Respond to SEND using memmapfile class.
```

```
disp('ANSWER server is awaiting message');

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:answer:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Wait till first byte is not zero.
    while m.Data(1) == 0
        pause(.25);
    end

    % The first byte now contains the length of the message.
    % Get it from m.
    msg = char(m.Data(2:1+m.Data(1)))';

    % Display the message.
    disp('Received message from SEND:')
    disp(msg)

    % Transform the message to all uppercase.
    m.Data(2:1+m.Data(1)) = upper(msg);

    % Signal to SEND that the response is ready.
    m.Data(1) = 0;
end
```

Running the Demo

Here is what the demonstration looks like when it is run. First, start two separate MATLAB sessions on the same computer system. Call the send function in one and the answer function in the other to create a map in each of the processes' memory to the common file:

```
% Run SEND in the first MATLAB session.
send
Enter send string (or RETURN to end):


% Run ANSWER in the second MATLAB session.
answer
ANSWER server is awaiting message
```

Next, enter a message at the prompt displayed by the send function. MATLAB writes the message to the shared file. The second MATLAB session, running the answer function, loops on byte 1 of the shared file and, when the byte is written by send, answer reads the message from the file via its memory map. The answer function then puts the message into uppercase and writes it back to the file, and send (waiting for a reply) reads the message and displays it:

```
% SEND writes a message and reads the uppercase reply.
Hello. Is there anybody out there?
response from ANSWER is:
HELLO.  IS THERE ANYBODY OUT THERE?
Enter send string (or RETURN to end):


% ANSWER reads the message from SEND.
Received message from SEND:
Hello.  Is there anybody out there?
```

send writes a second message to the file. answer reads it, put it into uppercase, and then writes the message to the file:

```
% SEND writes a second message to the shared file.
I received your reply.
response from ANSWER is:
I RECEIVED YOUR REPLY.
Enter send string (or RETURN to end): <Enter>
```

```
Terminating SEND function.


% ANSWER reads the second message.
Received message from SEND:
I received your reply.
```

Exporting Data to MAT-Files

In this section...

"MAT-Files" on page 6-64

"Using the save Function" on page 6-64

"Saving Structures" on page 6-65

"Appending to an Existing File" on page 6-66

"Data Compression" on page 6-66

"Unicode Character Encoding" on page 6-68

"Optional Output Formats" on page 6-69

"Storage Requirements" on page 6-70

"Saving From External Programs" on page 6-71

MAT-Files

MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

This section explains how to save the variables in your MATLAB session to a binary file called a MAT-file. The next section explains how to load them back into your MATLAB workspace.

Using the save Function

To export workspace variables to a binary or ASCII file, use the `save` function. You can save all variables from the workspace in a single operation (if you omit the filename, MATLAB uses the name `matlab.mat`):

```
save filename
```

or save just those variables that you specify:

```
save filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to save those variables that match a specific pattern. For example, the following command saves all variables that start with `str`.

```
save strinfo str*
```

Use `whos -file` to examine what has been written to the MAT-file:

```
whos -file strinfo
  Name           Size                    Bytes  Class

  str2           1x15                       30  char array
  strarray       2x5                       678  cell array
  strlen         1x1                         8  double array
```

Saving Structures

When saving a MATLAB structure, you have the option of saving the entire structure, saving each structure field as an individual variable in the MAT-file, or saving specific fields as individual variables.

For structure `S`,

```
S.a = 12.7;  S.b = {'abc', [4 5; 6 7]};  S.c = 'Hello!';
```

Save the entire structure to `newstruct.mat` with the usual syntax:

```
save newstruct.mat S;

whos -file newstruct
  Name      Size                   Bytes  Class

  S         1x1                      550  struct array
```

Save the fields individually with the `-struct` option:

```
save newstruct.mat -struct S;

whos -file newstruct
```

```
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  b         1x2                       158  cell array
  c         1x6                        12  char array
```

Or save only selected fields using `-struct` and specifying each field name:

```
save newstruct.mat -struct S a c;

whos -file newstruct
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  c         1x6                        12  char array
```

Appending to an Existing File

You can add new variables to those already stored in an existing MAT-file by using `save -append`. When you append to a MAT-file, MATLAB first looks in the designated file for each variable name specified in the argument list, or for all variables if no specific variable names are specified. Based on that information, MATLAB does both of the following:

- For each variable that already exists in the MAT-file, MATLAB overwrites its saved value with the new value taken from the workspace.
- For each variable not found in the MAT-file, MATLAB adds that variable to the file and stores its value from the workspace.

Note Saving with the `-append` option does not append additional elements to any arrays that are already saved in the MAT-file.

Data Compression

MATLAB compresses the data that you save to a MAT-file. Data compression can save you a significant amount of storage space when you are working with large files or working over a network.

Data compression is optional, however, and you can disable it either for an individual `save` operation, or for all of your MATLAB sessions. Use the `-v6` option with the `save` function to turn off compression on a per-command basis:

```
save filename -v6
```

To disable data compression for all of your MATLAB sessions, open the **Preferences** dialog, select **General** and then **MAT-Files**, and click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

Note You cannot read a compressed MAT-file with MATLAB versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with data compression disabled.

Information returned by the command `whos -file` is independent of whether the variables in that file are compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to.

Evaluating When to Compress

You should consider both data set size and the type of data being saved when deciding whether or not to compress the data you save to a file. The benefits of data compression are greater when saving large data sets (over 3 MB), and are usually negligible with smaller data sets. Data that has repeating patterns or more consistent values compresses better than random data. Compressing data that has a random pattern is not recommended as it slows down the performance of `save` and `load` significantly, and offers little benefit in return.

In general, data compression and decompression slows down all `save` and some `load` operations to some extent. In most cases, however, the resulting reduction in file size is worth the additional time spent compressing or decompressing. Because loading is typically done more frequently than saving, `load` is considered to be the most critical of the two operations. Up to a certain threshold (relative to the size of the uncompressed MAT-file), loading a compressed MAT-File is slightly slower than loading an uncompressed

file containing the same data. Beyond that threshold, however, loading the compressed file is *faster*.

For example, say that you have a block of data that takes up 100 MB in memory, and this data has been saved to both a 10 MB compressed file and a 100 MB uncompressed file. When you load each of these files back into the MATLAB workspace, the first 10 MB of data takes the same amount of time to load for each file. Loading the remaining 90 MB from the uncompressed file will take 9 times as long as the first 10 MB, while all that remains to be done with the compressed file is to decompress the data, and this takes a relatively short amount of time.

The loading size threshold is lower for network files, and also varies depending on the type of computer being used. Network users loading compressed MAT-files generally see faster load times than when loading uncompressed files, and at smaller data sizes than users loading the same files locally.

Note Compression and decompression during `save` and `load` is done transparently without the use of temporary files on disk. This is of significance to large dataset users in particular.

Unicode Character Encoding

MATLAB saves character data to a MAT-file using Unicode character encoding. As with data compression, Unicode character encoding is optional. If you disable it, MATLAB writes the MAT-file using the default encoding for your system. To disable Unicode character encoding on a per-command basis, use the `-v6` option with the `save` function:

```
save filename -v6
```

To disable Unicode character encoding for all of your MATLAB sessions, open the **Preferences** dialog, select **General** and then **MAT-Files**, and click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

When writing character data to a non-HDF5-based MAT-file using Unicode encoding (the default), MATLAB checks if the data is 7-bit ASCII. If it is,

MATLAB writes the 7-bit ASCII character data to the MAT-file using 8 bits per character (UTF-8 format), thus minimizing the size of the resulting file. Any character data that is not 7-bit ASCII is written in 16-bit Unicode form (UTF-16). This algorithm operates on a per-string basis.

Note You cannot read a Unicode encoded MAT-file with MATLAB versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with Unicode character encoding disabled.

For more information on how MATLAB saves specific ASCII data formats, and on preventing loss or corruption of character data, see "Writing Character Data" in the MATLAB External Interfaces documentation.

Optional Output Formats

You can choose from any of the following formats for your output file. If you do not specify a format, MATLAB uses the binary MAT-file format.

Output File Format	Command
Binary MAT-file (default)	`save` *`filename`*
8-digit ASCII	`save` *`filename`* `-ascii`
8-digit ASCII, tab delimited	`save` *`filename`* `-ascii -tabs`
16-digit ASCII	`save` *`filename`* `-ascii -double`
16-digit ASCII, tab delimited	`save` *`filename`* `-ascii -double -tabs`
MATLAB Version 4 compatible	`save` *`filename`* `-v4`

Saving in ASCII Format

When saving in any of the ASCII formats, consider the following:

- Each variable to be saved must be either a two-dimensional double array or a two-dimensional character array. Saving a complex double array causes the imaginary part of the data to be lost, as MATLAB cannot load nonnumeric data (`'i'`).

- To read the file with the MATLAB `load` function, make sure all the variables have the same number of columns. If you are using a program other than MATLAB to read the saved data, this restriction can be relaxed.
- Each MATLAB character in a character array is converted to a floating-point number equal to its internal ASCII code and written out as a floating-point number string. There is no information in the saved file that indicates whether the value was originally a number or a character.
- The values of all variables saved merge into a single variable that takes the name of the ASCII file (minus any extension). Therefore, it is advisable to save only one variable at a time.

Saving in Version 4 Format

With the `-v4` option, you can save only those data constructs that are compatible with MATLAB Version 4. Therefore, you cannot save structures, cell arrays, multidimensional arrays, or objects. Variable names cannot exceed 19 characters in length. In addition, you must use filenames that are supported by MATLAB Version 4.

Storage Requirements

The binary formats used by `save` depend on the size and type of each array. Arrays with any noninteger entries and arrays with 10,000 or fewer elements are saved in floating-point formats requiring 8 bytes per real element. Arrays with all integer entries and more than 10,000 elements are saved in the formats shown, requiring fewer bytes per element.

Element Range	Bytes per Element
0 to 255	1
0 to 65535	2
-32767 to 32767	2
-2^{31} to $2^{31}-1$	4
Other	8

Saving From External Programs

The *MATLAB External Interfaces* documentation provides details on reading and writing MAT-files from external C or Fortran programs. It is important to use recommended access methods, rather than rely upon the specific MAT-file format, which is likely to change in the future.

Importing Data From MAT-Files

In this section...

"Using the load Function" on page 6-72

"Previewing MAT-File Contents" on page 6-72

"Loading Into a Structure" on page 6-73

"Loading Binary Data" on page 6-73

"Loading ASCII Data" on page 6-74

Using the load Function

To import variables from a binary or ASCII file on your disk to your workspace, use the `load` function. You can load all variables from the workspace in a single operation (if you omit the filename, MATLAB loads from file `matlab.mat`):

```
load filename
```

or load just those variables that you specify:

```
load filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to load those variables that match a specific pattern. (This works for MAT-files only.) For example, the following command loads all variables that start with `str` from file `strinfo.mat`:

```
load strinfo str*
```

Caution When you import data into the MATLAB workspace, it overwrites any existing variable in the workspace with the same name.

Previewing MAT-File Contents

To see what variables are stored in a MAT-file before actually loading the file into your workspace, use `whos -file` *`filename`*. This command returns the name, dimensions, size, and data type of all variables in the specified MAT-file.

You can use whos -file on binary MAT-files only:

```
whos -file mydata.mat
  Name           Size             Bytes  Class

  javArray      10x1                       java.lang.Double[][]
  spArray        5x5                  84  double array (sparse)
  strArray       2x5                 678  cell array
  x              3x2x2                96  double array
  y              4x5                1230  cell array
```

Loading Into a Structure

To load MAT-file data into a MATLAB structure, specify an output variable in your load command. This example reads the data in mydata.mat into the fields of structure S:

```
S = load('mydata.mat')
S =
           x: [3x2x2 double]
           y: {4x5 cell}
     spArray: [5x5 double]
    strArray: {2x5 cell}
    javArray: [10x1 java.lang.Double[][]]

whos S
  Name      Size                    Bytes  Class

  S         1x1                      2840  struct array
```

Loading Binary Data

MAT-files are double-precision binary MATLAB format files created by the save function and readable by the load function. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs, external to MATLAB.

MAT-files can contain data in an uncompressed or a compressed form, or both. MATLAB knows which variables in the file have been compressed by looking at a tag that it attaches to each variable during the save operation. When

loading data from a MAT-file into the workspace, MATLAB automatically handles the decompression of the appropriate data.

The External Interface libraries contain C- and Fortran-callable routines to read and write MAT-files from external programs.

Loading ASCII Data

ASCII files must be organized as a rectangular table of numbers, with each number in a row separated by a blank, comma, or tab character, and with an equal number of elements in each row. MATLAB generates an error if the number of values differs between any two rows. ASCII files can contain MATLAB comments (lines that begin with `%`).

MATLAB returns all the data in the file as a single two-dimensional array of type `double`. The number of rows in the array is equal to the number of lines in the file, and the number of columns is equal to the number of values on a line.

In the workspace, MATLAB assigns the array to a variable named after the file being loaded (minus any file extension). For example, the command

```
load mydata.dat
```

reads all of the data from `mydata.dat` into the MATLAB workspace as a single array, and assigns it to a variable called `mydata`. In naming the variable, `load` precedes any leading underscores or digits in `filename` with an `X` and replaces any other nonalphabetic characters with underscores.

For example, the command

```
load 10-May-data.dat
```

assigns the data in file `10-May-data.dat` to a new workspace variable called `X10_May_data`.

Importing Text Data

In this section...

"The MATLAB Import Wizard" on page 6-75
"Using Import Functions with Text Data" on page 6-75
"Importing Numeric Text Data" on page 6-78
"Importing Delimited ASCII Data Files" on page 6-79
"Importing Numeric Data with Text Headers" on page 6-80
"Importing Mixed Alphabetic and Numeric Data" on page 6-81
"Importing from XML Documents" on page 6-83

Caution When you import data into the MATLAB workspace, you overwrite any existing variable in the workspace with the same name.

The MATLAB Import Wizard

The easiest way to import data into MATLAB is to use the Import Wizard. You do not need to know the format of the data to use this tool. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically.

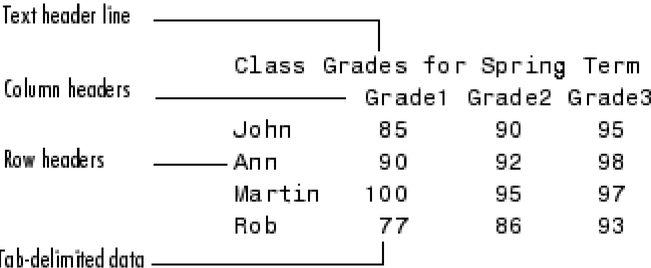
For more information, see "Using the Import Wizard" on page 6-11.

Using Import Functions with Text Data

To import text data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

The text data must be formatted in a uniform pattern of rows and columns, using a text character, called a *delimiter* or *column separator*, to separate each data item. The delimiter can be a space, comma, semicolon, tab, or any other character. The individual data items can be alphabetic or numeric characters or a mix of both.

The text file can also contain one or more lines of text, called *header lines*, or can use text headers to label each column or row. The following example illustrates a tab-delimited text file with header text and row and column headers.



To find out how your data is formatted, view it in a text editor. After you determine the format, find the sample in the table below that most closely resembles the format of your data. Then read the topic referred to in the table for information on how to import that format.

Table 6-1 ASCII Data File Formats

Data Format Sample	File Extension	Description
`1 2 3 4 5` `6 7 8 9 10`	.txt .dat or other	See "Importing Numeric Text Data" on page 6-78 or "Using the Import Wizard" on page 6-11 for information.

`1; 2; 3; 4; 5`
`6; 7; 8; 9; 10`
`or`
`1, 2, 3, 4, 5`
`6, 7, 8, 9, 10`
	.txt
	.dat
	.csv
	or other
		See "Importing Delimited ASCII Data Files" on page 6-79 or "Using the Import Wizard" on page 6-11 for information.

Table 6-1 ASCII Data File Formats (Continued)

Data Format Sample	File Extension	Description
`Ann Type1 12.34 45 Yes` `Joe Type2 45.67 67 No`	.txt .dat or other	See "Importing Mixed Alphabetic and Numeric Data" on page 6-81 for information.

`Grade1 Grade2 Grade3`
`91.5   89.2   77.3`
`88.0   67.8   91.0`
`67.3    78.1   92.5`
	.txt
	.dat
	or other
		See "Importing Numeric Data with Text Headers" on page 6-80 or "Using the Import Wizard" on page 6-11 for information.

If you are familiar with MATLAB import functions but are not sure when to use them, see the following table, which compares the features of each function.

Table 6-2 ASCII Data Import Function Features

Function	Data Type	Delimiters	Number of Return Values	Notes
`csvread`	Numeric data	Commas only	One	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-98.
`dlmread`	Numeric data	Any character	One	Flexible and easy to use.

Table 6-2 ASCII Data Import Function Features (Continued)

Function	Data Type	Delimiters	Number of Return Values	Notes
`fscanf`	Alphabetic and numeric; however, both types returned in a single return variable	Any character	One	Part of low-level file I/O routines. Requires use of `fopen` to obtain file identifier and `fclose` after read.
`load`	Numeric data	Spaces only	One	Easy to use. Use the functional form of `load` to specify the name of the output variable.
`textread`	Alphabetic and numeric	Any character	Multiple values in cell arrays	Flexible, powerful, and easy to use. Use format string to specify conversions.
`textscan`	Alphabetic and numeric	Any character	Multiple values returned to one cell array	More flexible than `textread`. Also more format options.

Importing Numeric Text Data

If your data file contains only numeric data, you can use many of the MATLAB import functions (listed in ASCII Data Import Function Features on page 6-77), depending on how the data is delimited. If the data is rectangular, that is, each row has the same number of elements, the simplest command to use is the `load` command. (The `load` function can also be used to import MAT-files, the MATLAB binary format for saving the workspace.)

For example, the file named my_data.txt contains two rows of numbers delimited by space characters:

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

When you use load as a command, it imports the data and creates a variable in the workspace with the same name as the filename, minus the file extension:

```
load my_data.txt;
whos
   Name         Size         Bytes   Class

   my_data      2x5            80    double array

my_data

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

If you want to name the workspace variable something other than the filename, use the functional form of load. In the following example, the data from my_data.txt is loaded into the workspace variable A:

```
A = load('my_data.txt');
```

Importing Delimited ASCII Data Files

If your data file uses a character other than a space as a delimiter, you have a choice of several import functions you can use. (See ASCII Data Import Function Features on page 6-77 for a complete list.) The simplest to use is the dlmread function.

For example, consider a file named ph.dat whose contents are separated by semicolons:

```
7.2;8.5;6.2;6.6
5.4;9.2;8.1;7.2
```

To read the entire contents of this file into an array named A, enter

```
A = dlmread('ph.dat', ';');
```

You specify the delimiter used in the data file as the second argument to dlmread. Note that, even though the last items in each row are not followed by a delimiter, dlmread can still process the file correctly. dlmread ignores space characters between data elements. So, the preceding dlmread command works even if the contents of ph.dat are

```
7.2;   8.5;       6.2;6.6
5.4;   9.2   ;8.1;7.2
```

Importing Numeric Data with Text Headers

To import an ASCII data file that contains text headers, use the textscan function, specifying the headerlines parameter. textscan accepts a set of predefined parameters that control various aspects of the conversion. (For a complete list of these parameters, see the textscan reference page.) Using the headerlines parameter, you can specify the number of lines at the head of the file that textscan should ignore.

For example, the file grades.dat contains formatted numeric data with a one-line text header:

```
Grade1 Grade2 Grade3
  78.8   55.9   45.9
  99.5   66.8   78.0
  89.5   77.0   56.7
```

To import this data, first open the file and then use this textscan command to read the contents:

```
fid = fopen('grades.dat', 'r');
grades = textscan(fid, '%f %f %f', 3, 'headerlines', 1);

grades{:}
ans =
   78.8000
   99.5000
   89.5000

ans =
   55.9000
```

```
    66.8000
    77.0000

ans =
    45.9000
    78.0000
    56.7000

fclose(fid);
```

Importing Mixed Alphabetic and Numeric Data

If your data file contains a mix of alphabetic and numeric ASCII data, use the `textscan` or `textread` function to import the data. `textscan` returns its output in a single cell array, while `textread` returns its output in separate variables and you can specify the data type of each variable. The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.

This example uses `textread` to import the file `mydata.dat` that contains a mix of alphabetic and numeric data:

```
Sally    Type1 12.34 45 Yes
Larry    Type2 34.56 54 Yes
Tommy    Type1 67.89 23 No
```

Note To read an ASCII data file that contains numeric data with text column headers, see “Importing Numeric Data with Text Headers” on page 6-80.

To read the entire contents of the file `mydata.dat` into the workspace, specify the name of the data file and the format string as arguments to `textread`. In the format string, you include conversion specifiers that define how you want each data item to be interpreted. For example, specify `%s` for string data, `%f` for floating-point data, and so on. (For a complete list of format specifiers, see the `textread` reference page.)

For each conversion specifier in your format string, you must specify a separate output variable. `textread` processes each data item in the file as specified in the format string and puts the value in the output variable. The

number of output variables must match the number of conversion specifiers in the format string.

In this example, `textread` reads the file `mydata.dat`, applying the format string to each line in the file until the end of the file:

```
[names, types, x, y, answer] = ...
   textread('mydata.dat', '%s %s %f %d %s', 3)
names =
    'Sally'
    'Larry'
    'Tommy'

types =
    'Type1'
    'Type2'
    'Type1'

x =
   12.3400
   34.5600
   67.8900

y =
    45
    54
    23

answer =
    'Yes'
    'Yes'
    'No'
```

If your data uses a character other than a space as a delimiter, you must use the `textread` parameter `'delimiter'` to specify the delimiter. For example, if the file `mydata.dat` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer]= ...
   textread('mydata.dat', '%s %s %f %d %s', 'delimiter', ';')
```

See the `textread` reference page for more information about these optional parameters.

Importing from XML Documents

With the `xmlread` function, you can read from a given URL or file, generating a Document Object Model (DOM) node to represent the parsed document.

MATLAB also provides these other XML functions:

- `xmlwrite` — Serializes a Document Object Model node to a file
- `xslt` — Transforms an XML document using an XSLT engine

See the reference pages for these functions for more information.

Exporting Text Data

In this section...

"Overview" on page 6-84
"Exporting Delimited ASCII Data Files" on page 6-86
"Using the diary Function to Export Data" on page 6-87
"Exporting to XML Documents" on page 6-88

Overview

This section describes how to use MATLAB functions to export data in several common ASCII formats. For example, you can use these functions to export a MATLAB matrix as a text file where the rows and columns are represented as space-separated, numeric values. The function you use depends on the amount of data you want to export and its format. Topics covered include

If you are not sure which section describes your data, find the sample in the table below that most nearly matches the data format you want to create. Then read the section referred to in the table.

If you are familiar with MATLAB export functions but are not sure when to use them, see ASCII Data Export Function Features on page 6-85, which compares the features of each function.

Note If C or Fortran routines for writing data files in the form needed by other applications exist, create a MEX-file to write the data. See the *MATLAB External Interfaces* documentation for more information.

Table 6-3 ASCII Data File Formats

Data Format Sample	MATLAB Export Function
`1 2 3 4 5 6` `7 8 9 10`	See "Exporting Delimited ASCII Data Files" on page 6-86 and "Using the diary Function to Export Data" on page 6-87 for information about these options.

`1; 2; 3; 4;`
`5; 6; 7; 8;`
`9; 10;`
	See "Exporting Delimited ASCII Data Files" on page 6-86 for information. The example shows a semicolon-delimited file, but you can specify another character as the delimiter.

Table 6-4 ASCII Data Export Function Features

Function	Use With	Delimiters	Notes
`csvwrite`	Numeric data	Commas only	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-98.
`diary`	Numeric data or cell array	Spaces only	Can be used for small arrays. Requires editing of data file to remove extraneous text.
`dlmwrite`	Numeric data	Any character	Easy to use, flexible.

Table 6-4 ASCII Data Export Function Features (Continued)

Function	Use With	Delimiters	Notes
`fprintf`	Alphabetic and numeric data	Any character	Part of low-level file I/O routines. This function is the most flexible but also the most difficult to use. You must use `fopen` to obtain a file identifier before writing the data and `fclose` to close the file after writing the data.
`save`	Numeric data	Tabs or spaces	Easy to use; output values are high precision.

Exporting Delimited ASCII Data Files

To export an array as a delimited ASCII data file, you can use either the `save` function, specifying the `-ASCII` qualifier, or the `dlmwrite` function. The `save` function is easy to use; however, the `dlmwrite` function provides more flexibility, allowing you to specify any character as a delimiter and to export subsets of an array by specifying a range of values.

Using the save Function

To export the array `A`,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

use the `save` function, as follows:

```
save my_data.out A -ASCII
```

If you view the created file in a text editor, it looks like this:

```
1.0000000e+000   2.0000000e+000   3.0000000e+000   4.0000000e+000
5.0000000e+000   6.0000000e+000   7.0000000e+000   8.0000000e+000
```

By default, `save` uses spaces as delimiters but you can use tabs instead of spaces by specifying the `-tabs` option.

When you use save to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. If you write the character string 'hello' to a file, save writes the values

```
104 101 108 108 111
```

Using the dlmwrite Function

To export an array in ASCII format and specify the delimiter used in the file, use the dlmwrite function.

For example, to export the array A,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

as an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

If you view the created file in a text editor, it looks like this:

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

Note that dlmwrite does not insert delimiters at the end of rows.

By default, if you do not specify a delimiter, dlmwrite uses a comma as a delimiter. You can specify a space (' ') as a delimiter or, if you specify empty quotes (''), no delimiter.

Using the diary Function to Export Data

To export small numeric arrays or cell arrays, you can use the diary function. diary creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array A in your workspace,

```
A = [ 1 2 3 4; 5 6 7 8 ];
```

execute these commands at the MATLAB prompt to export this array using `diary`:

1 Turn on the `diary` function. You can optionally name the output file `diary` creates.

```
diary my_data.out
```

2 Display the contents of the array you want to export. This example displays the array `A`. You could also display a cell array or other MATLAB data type.

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

3 Turn off the `diary` function.

```
diary off
```

`diary` creates the file `my_data.out` and records all the commands executed in the MATLAB session until it is turned off.

```
A =

      1     2     3     4
      5     6     7     8

diary off
```

4 Open the `diary` file `my_data.out` in a text editor and remove all the extraneous text.

Exporting to XML Documents

With the `xmlwrite` function, you can serialize a Document Object Model (DOM) node to an XML file.

MATLAB also provides these other XML functions:

- `xmlread` — Imports from a given URL or file to a Document Object Model node

- `xslt` — Transforms an XML document using an XSLT engine

See the reference pages for these functions for more information.

Working with Graphics Files

In this section...

"Getting Information About Graphics Files" on page 6-90

"Importing Graphics Data" on page 6-91

"Exporting Graphics Data" on page 6-91

Getting Information About Graphics Files

If you have a file in a standard graphics format, use the `imfinfo` function to get information about its contents. The `imfinfo` function returns a structure containing information about the file. The fields in the structure vary with the file format but `imfinfo` always returns some basic information including filename, last modification date, file size, and format.

This example returns information about a file in Joint Photographic Experts Group (JPEG) format:

```
info = imfinfo('ngc6543a.jpg')

info =

           Filename: [1x57 char]
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {[1x69 char]}
```

Importing Graphics Data

To import data into the MATLAB workspace from a graphics file, use the imread function. Using this function, you can import data from files in many standard file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats. For a complete list of supported formats, see the imread reference page.

This example reads the image data stored in a file in JPEG format into the MATLAB workspace as the array I:

```
I = imread('ngc6543a.jpg');
```

imread represents the image in the workspace as a multidimensional array of class uint8. The dimensions of the array depend on the format of the data. For example, imread uses three dimensions to represent RGB color images:

```
whos I
  Name      Size                             Bytes  Class

  I       650x600x3                        1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
```

Exporting Graphics Data

To export data from the MATLAB workspace using one of the standard graphics file formats, use the imwrite function. Using this function, you can export data in formats such as the Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG). For a complete list of supported formats, see the imwrite reference page.

The following example writes a multidimensional array of uint8 data I from the MATLAB workspace into a file in TIFF format. The class of the output image written to the file depends on the format specified. For most formats, if the input array is of class uint8, imwrite outputs the data as 8-bit values. See the imwrite reference page for details.

```
whos I
  Name      Size                             Bytes  Class
```

```
  I       650x600x3                       1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
imwrite(I, 'my_graphics_file.tif','tif');
```

Note `imwrite` supports different syntaxes for several of the standard formats. For example, with TIFF file format, you can specify the type of compression used to store the image. See the `imwrite` reference page for details.

Working with Audio and Video Data

In this section...

"Getting Information About Audio/Video Files" on page 6-93

"Importing Audio/Video Data" on page 6-94

"Exporting Audio/Video Data" on page 6-95

Getting Information About Audio/Video Files

MATLAB includes several functions that you can use to get information about files that contain audio data, video data, or both. Some work only with specific file formats. One function, the `mmfileinfo` function, can retrieve information about many file formats.

Format-Specific Functions

MATLAB includes several functions that return information about files that contain audio and video data in specific formats.

- `aufinfo` — Returns a text description of the contents of a sound (AU) file
- `aviinfo` — Returns a structure containing information about the contents of an Audio/Video Interleaved (AVI) file
- `wavfinfo` — Returns a text description of the contents of a sound (WAV) file

Using the General Multimedia Information Function

MATLAB also includes a general-purpose, audio/video file information function named `mmfileinfo`. The `mmfileinfo` function returns information about both the audio data in a file as well as the video data in the file, if present.

Note `mmfileinfo` can be used only on Windows systems.

Importing Audio/Video Data

MATLAB includes several functions that you can use to bring audio or video data into the MATLAB workspace. Some of these functions read audio or video data from files. Another way to import audio data into the MATLAB workspace is to record it using an audio input device, such as a microphone. The following sections describe

- "Reading Audio and Video Data from a File" on page 6-94
- "Recording Audio Data" on page 6-94

Reading Audio and Video Data from a File

MATLAB includes several functions for reading audio or video data from a file. These files are format-specific.

- `auread` — Returns sound data from a sound (AU) file
- `aviread` — Returns AVI data as a MATLAB movie
- `mmreader` — Returns AVI, MPG, or WMV video data
- `wavread` — Returns sound data from a sound (WAV) file

Note `mmreader` can be used only on Windows systems.

Recording Audio Data

To bring sound data into the MATLAB workspace by recording it from an audio input device, use the audio recorder object. This object represents the connection between MATLAB and an audio input device, such as a microphone, that is connected to your system. You use the `audiorecorder` function to create this object and then use methods and properties of the object to record the audio data.

On PCs running Windows, you can also use the `wavrecord` function to bring live audio data in WAV format into the MATLAB workspace.

Once you import audio data, MATLAB supports several ways to listen to the data. You can use an audio player object to play the audio data. Use the `audioplayer` function to create an audio player object.

You can also use the `sound` or `soundsc` function.

On PCs running Windows, you can use the `wavplay` function to listen to `.wav` files.

Exporting Audio/Video Data

MATLAB includes several functions that you can use to export audio or video data from the MATLAB workspace. These functions write audio data to a file using specific file formats. The following sections describe

- "Exporting Audio Data" on page 6-95
- "Exporting Video Data in AVI Format" on page 6-95

This section also provides an example of writing video data to a file in "Example: Creating an AVI file" on page 6-96.

Exporting Audio Data

In MATLAB, audio data is simply numeric data that you can export using standard MATLAB data export functions, such as `save`.

MATLAB also includes several functions that write audio data to files in specific file formats:

- `auwrite` — Exports sound data in AU file format
- `wavwrite` — Exports sound data in WAV file format

Exporting Video Data in AVI Format

You can export MATLAB video data as an Audio/Video Interleaved (AVI) file. To do this, you use the `avifile` function to create an `avifile` object. Once you have the object, you can use AVI file object methods and properties to control various aspects of the data export process.

For example, in MATLAB, you can save a sequence of graphs as a movie that can then be played back using the `movie` function. You can export a MATLAB movie by saving it in MAT-file format, like any other MATLAB workspace variable. However, anyone who wants to view your movie must have MATLAB. (For more information about MATLAB movies, see the Animation section in the MATLAB Graphics documentation.)

To export a sequence of MATLAB graphs in a format that does not require MATLAB for viewing, save the figures in Audio/Video Interleaved (AVI) format. AVI is a file format that allows animation and video clips to be played on a PC running Windows or on UNIX systems.

Note To convert an existing MATLAB movie into an AVI file, use the `movie2avi` function.

Example: Creating an AVI file

To export a sequence of MATLAB graphs as an AVI format movie, perform these steps:

1. Create an AVI file object, using the `avifile` function.

   ```
   aviobj = avifile('mymovie.avi','fps',5);
   ```

   AVI file objects support properties that let you control various characteristics of the AVI movie, such as colormap, compression, and quality. (See the `avifile` reference page for a complete list.) `avifile` uses default values for all properties, unless you specify a value. The example sets the value of the frames per second (`fps`) property.

2. Capture the sequence of graphs and put them into the AVI file, using the `addframe` function.

   ```
   for k=1:25
       h = plot(fft(eye(k+16)));
       set(h,'EraseMode','xor');
       axis equal;
       frame = getframe(gca);
       aviobj = addframe(aviobj,frame);
   ```

```
end
```

The example uses a `for` loop to capture the series of graphs to be included in the movie. You typically use `addframe` to capture a sequence of graphs for AVI movies. However, because this particular MATLAB animation uses `XOR` graphics, you must call `getframe` to capture the graphs and then call `addframe` to add the captured frame to the movie.

3 Close the AVI file, using the `close` function.

```
aviobj = close(aviobj);
```

Working with Spreadsheets

In this section...

"Microsoft Excel Spreadsheets" on page 6-98

"Lotus 123 Spreadsheets" on page 6-101

Microsoft Excel Spreadsheets

This section covers

- "Getting Information About the File" on page 6-98
- "Exporting to the File" on page 6-99
- "Importing from the File" on page 6-100

See the `xlsfinfo`, `xlswrite`, and `xlsread` reference pages for more detailed information and examples.

Getting Information About the File

Use the `xlsfinfo` function to determine if a file contains a readable Microsoft Excel spreadsheet.

Inputs to `xlsfinfo` are

- Name of the spreadsheet file

Outputs from `xlsfinfo` are

- String `'Microsoft Excel Spreadsheet'` if the file contains an Excel worksheet readable with the `xlsread` function. Otherwise, it contains an empty string (`''`).
- Cell array of strings containing the names of each worksheet in the file.

Example — Querying an XLS File. This example returns information about spreadsheet file `tempdata.xls`:

```
[type, sheets] = xlsfinfo('tempdata.xls')
```

```
type =
Microsoft Excel Spreadsheet
sheets =
    'Locations'    'Rainfall'    'Temperatures'
```

Exporting to the File

Use the xlswrite function to export a matrix to an Excel spreadsheet file. With xlswrite, you can export data from the workspace to any worksheet in the file, and to any location within that worksheet.

Inputs to xlswrite are

- Name of the spreadsheet file
- Matrix to be exported
- Name of the worksheet to receive the data
- Range of cells on the worksheet in which to write the data

Outputs from xlswrite are

- Pass or fail status
- Any warning or error message generated along with its message identifier

Example — Writing To an XLS File. This example writes a mix of text and numeric data to the file tempdata.xls. Call xlswrite, specifying a worksheet labeled Temperatures, and the region within the worksheet to write the data to. The 4-by-2 matrix is written to the rectangular region that starts at cell E1 in its upper-left corner:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97}
d =
    'Time'    'Temp'
    [  12]    [  98]
    [  13]    [  99]
    [  14]    [  97]

xlswrite('tempdata.xls', d, 'Temperatures', 'E1');
```

Adding a New Worksheet. If the worksheet being written to does not already exist in the file, MATLAB displays the following warning:

```
Warning: Added specified worksheet.
```

You can disable these warnings with the command

```
warning off MATLAB:xlswrite:AddSheet
```

Importing from the File

Use `xlsread` to import a matrix from an Excel spreadsheet file into the MATLAB workspace. You can import data from any worksheet in the file, and from any location within that worksheet. You can also optionally have `xlsread` open an Excel window showing the file and then interactively select the worksheet and range of data to be read by the function.

Inputs to `xlsread` are

- Name of the spreadsheet file
- Matrix to be imported
- Name of the worksheet from which to read the data
- Range of cells on the worksheet from which to read the data
- Keyword that opens an Excel window, enabling you to interactively select the worksheet and range of data to read
- Keyword that imports using basic import mode

Three separate outputs from `xlsread` are

- Numeric data
- String data
- Any unprocessed cell content

Example — Reading from an XLS File. Continuing with the previous example, to import only the numeric data, use `xlsread` with a single return argument. `xlsread` ignores any leading row or column of text in the numeric result:

```
ndata = xlsread('tempdata.xls', 'Temperatures')
ndata =
    12    98
    13    99
    14    97
```

To import both numeric data and text data, specify two return values for `xlsread`:

```
[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')

headertext =
    'Time'    'Temp'

ndata =
    12    98
    13    99
    14    97
```

Lotus 123 Spreadsheets

This section covers

- "Getting Information About the File" on page 6-101
- "Exporting to the File" on page 6-102
- "Importing from the File" on page 6-103

See the `wk1finfo`, `wk1write`, and `wk1read` reference pages for more detailed information and examples.

Getting Information About the File

Use the `wk1finfo` function to determine if a file contains a Lotus WK1 spreadsheet:

Inputs to `wk1finfo` are

- Name of the spreadsheet file

Outputs from `wk1finfo` are

- String `'WK1'` if the file is a Lotus spreadsheet readable with the `wk1read` function. Otherwise, it contains an empty string (`''`).
- String `'Lotus 123 Spreadsheet'`

Example — Querying a WK1 File. This example returns information about spreadsheet file `matA.wk1`:

```
[extens, type] = wk1finfo('matA.wk1')

extens =
   WK1
type =
   Lotus 123 Spreadsheet
```

Exporting to the File

Use the `wk1write` function to export a matrix to a Lotus spreadsheet file. You have the choice of positioning the matrix starting at the first row and column of the spreadsheet, or at any other location in the file.

To export to a specific location in the file, use the second syntax, indicating a zero-based starting row and column.

Inputs to `wk1write` are

- Name of the spreadsheet file
- Matrix to be exported
- Location in the file in which to write the data

Example — Writing to a WK1 File. This example exports an 8-by-8 matrix to spreadsheet file `matA.wk1`:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78];
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
```

```
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58
    61    62    63    64    65    66    67    68
    71    72    73    74    75    76    77    78

wk1write('matA.wk1', A);
```

Importing from the File

To import data from the spreadsheet into the MATLAB workspace, use `wk1read`. There are three ways to call `wk1read`. The first two shown here are similar to `wk1write`. The third enables you to select a range of values from the spreadsheet. You can specify the `range` argument with a one-based vector, spreadsheet notation (e.g., `'A1..B7'`), or using a named range (e.g., `'Sales'`).

Inputs to `wk1read` are

- Name of the spreadsheet file
- Spreadsheet location from which to read the data
- Range of cells from which to read the data

Outputs from `wk1read` are

- Requested data from the spreadsheet

Example — Reading from a WK1 File. Read in a limited block of the spreadsheet data by specifying the upper-left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

Using Low-Level File I/O Functions

In this section...

"Overview" on page 6-104

"Opening Files" on page 6-105

"Reading Binary Data" on page 6-107

"Writing Binary Data" on page 6-109

"Controlling Position in a File" on page 6-109

"Reading Strings Line by Line from Text Files" on page 6-112

"Reading Formatted ASCII Data" on page 6-113

"Writing Formatted Text Files" on page 6-114

"Closing a File" on page 6-115

Overview

MATLAB includes a set of low-level file I/O functions that are based on the I/O functions of the ANSI Standard C Library. If you know C, you are probably familiar with these routines.

To read or write data, perform these steps:

1. Open the file, using `fopen`. `fopen` returns a file identifier that you use with all the other low-level file I/O routines.

2. Operate on the file.

   a. Read binary data, using `fread`.

   b. Write binary data, using `fwrite`.

   c. Read text strings from a file line-by-line, using `fgets` or `fgetl`.

   d. Read formatted ASCII data, using `fscanf`.

   e. Write formatted ASCII data, using `fprintf`.

3. Close the file, using `fclose`.

This section also describes how these functions affect the current position in the file where read or write operations happen and how you can change the position in the file.

Note While the MATLAB file I/O commands are modeled on the C language I/O routines, in some ways their behavior is different. For example, the `fread` function is *vectorized*; that is, it continues reading until it encounters a text string or the end of file. These sections, and the MATLAB reference pages for these functions, highlight any differences in behavior.

Opening Files

Before reading or writing a text or binary file, you must open it with the `fopen` command.

```
fid = fopen('filename','permission')
```

Specifying the Permission String

The `permission` string specifies the kind of access to the file you require. Possible `permission` strings include

- `r` for reading only
- `w` for writing only
- `a` for appending only
- `r+` for both reading and writing

Note Systems such as Microsoft Windows that distinguish between text and binary files might require additional characters in the permission string, such as `'rb'` to open a binary file for reading.

Using the Returned File Identifier (fid)

If successful, `fopen` returns a nonnegative integer, called a *file identifier* (`fid`). You pass this value as an argument to the other I/O functions to access

the open file. For example, this `fopen` statement opens the data file named `penny.dat` for reading:

```
fid = fopen('penny.dat','r')
```

If `fopen` fails, for example if you try to open a file that does not exist, `fopen`

- Assigns `-1` to the file identifier.
- Assigns an error message to an optional second output argument. Note that the error messages are system dependent and are not provided for all errors on all systems. The function `ferror` can also provide information about errors.

Test the file identifier each time you open a file in your code. For example, this code loops until a readable filename is entered:

```
fid=0;
while fid < 1
   filename=input('Open file: ', 's');
   [fid,message] = fopen(filename, 'r');
   if fid == -1
     disp(message)
   end
end
```

When you run this code, if you specify a file that doesn't exist, such as `nofile.mat`, at the `Open file:` prompt, the results are

```
Open file: nofile.mat
Sorry. No help in figuring out the problem . . .
```

If you specify a file that does exist, such as `goodfile.mat`, the code example returns the file identifier, `fid`, and exits the loop.

```
Open file: goodfile.mat
```

Opening Temporary Files and Directories

The `tempdir` and `tempname` functions assist in locating temporary data on your system.

Function	Purpose
`tempdir`	Get temporary directory name.
`tempname`	Get temporary filename.

Use these functions to create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary can mean only that the file is not backed up.

The `tempdir` function returns the name of the directory or folder that has been designated to hold temporary files on your system. For example, issuing `tempdir` on a UNIX system returns the `/tmp` directory.

MATLAB also provides a `tempname` function that returns a filename in the temporary directory. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first:

```
fid = fopen(tempname, 'w');
```

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

Reading Binary Data

The `fread` function reads all or part of a binary file (as specified by a file identifier) and stores it in a matrix. In its simplest form, it reads an entire file and interprets each byte of input as the next element of the matrix. For example, the following code reads the data from a file named `nickel.dat` into matrix `A`:

```
fid = fopen('nickel.dat','r');
A = fread(fid);
```

To echo the data to the screen after reading it, use `char` to display the contents of `A` as characters, transposing the data so it is displayed horizontally:

```
disp(char(A'))
```

The `char` function causes MATLAB to interpret the contents of `A` as characters instead of as numbers. Transposing `A` displays it in its more natural horizontal format.

Controlling the Number of Values Read

`fread` accepts an optional second argument that controls the number of values read (if unspecified, the default is the entire file). For example, this statement reads the first 100 data values of the file specified by `fid` into the column vector `A`.

```
A = fread(fid,100);
```

Replacing the number `100` with the matrix dimensions `[10 10]` reads the same 100 elements into a 10-by-10 array.

Controlling the Data Type of Each Value

An optional third argument to `fread` controls the data type of the input. The data type argument controls both the number of bits read for each value and the interpretation of those bits as character, integer, or floating-point values. MATLAB supports a wide range of precisions, which you can specify with MATLAB specific strings or their C or Fortran equivalents.

Some common precisions include

- `'char'` and `'uchar'` for signed and unsigned characters (usually 8 bits)
- `'short'` and `'long'` for short and long integers (usually 16 and 32 bits, respectively)
- `'float'` and `'double'` for single- and double-precision floating-point values (usually 32 and 64 bits, respectively)

Note The meaning of a given precision can vary across different hardware platforms. For example, a 'uchar' is not always 8 bits. fread also provides a number of more specific precisions, such as 'int8' and 'float32'. If in doubt, use precisions that are not platform dependent. See fread for a complete list of precisions.

For example, if fid refers to an open file containing single-precision floating-point values, then the following command reads the next 10 floating-point values into a column vector A:

```
A = fread(fid,10,'float');
```

Writing Binary Data

The fwrite function writes the elements of a matrix to a file in a specified numeric precision, returning the number of values written. For instance, these lines create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, each stored as 4-byte integers:

```
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
```

In this case, fwrite sets the count variable to 25 unless an error occurs, in which case the value is less.

Controlling Position in a File

Once you open a file with fopen, MATLAB maintains a file position indicator that specifies a particular location within a file. MATLAB uses the file position indicator to determine where in the file the next read or write operation will begin. The following sections describe how to

- Determine whether the file position indicator is at the end of the file
- Move to a specific location in the file
- Retrieve the current location of the file position indicator
- Reset the file position indicator to the beginning of the file

Setting and Querying the File Position

The fseek and ftell functions enable you to set and query the position in the file at which the next input or output operation takes place:

- The fseek function repositions the file position indicator, letting you skip over data or back up to an earlier part of the file.
- The ftell function gives the offset in bytes of the file position indicator for a specified file.

The syntax for fseek is

```
status = fseek(fid,offset,origin)
```

fid is the file identifier for the file. offset is a positive or negative offset value, specified in bytes. origin is one of the following strings that specify the location in the file from which to calculate the position.

'bof'	Beginning of file
'cof'	Current position in file
'eof'	End of file

Example of Using fseek And ftell

To see how fseek and ftell work, consider this short M-file:

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
status = fclose(fid);
```

This code writes out the numbers 1 through 5 to a binary file named five.bin. The call to fwrite specifies that each numerical element be stored as a short. Consequently, each number uses two storage bytes.

Now reopen five.bin for reading:

```
fid = fopen('five.bin','r');
```

This call to `fseek` moves the file position indicator forward 6 bytes from the beginning of the file:

```
status = fseek(fid,6,'bof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator								↑				

This call to `fread` reads whatever is at file positions 7 and 8 and stores it in variable `four`:

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`:

```
position = ftell(fid)

position =

     8
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator										↑		

This call to `fseek` moves the file position indicator back 4 bytes:

```
status = fseek(fid,-4,'cof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator						↑						

Calling `fread` again reads in the next value (3):

```
three = fread(fid,1,'short');
```

Reading Strings Line by Line from Text Files

MATLAB provides two functions, fgetl and fgets, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that fgets copies the newline character to the string vector but fgetl does not.

The following M-file function demonstrates a possible use of fgetl. This function uses fgetl to read an entire file one line at a time. For each line, the function determines whether an input literal string (literal) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0
   tline = fgetl(fid);
   matches = findstr(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
end
fclose(fid);
```

For example, consider the following input data file called badpoem:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string 'an' appears in this file, use litcount:

```
litcount('badpoem','an')
2: Oranges and lemons,
```

```
1: Pineapples and tea.
3: Orangutans and monkeys,
```

Reading Formatted ASCII Data

The fscanf function is like the fscanf function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers for fscanf begin with a % character; common conversion specifiers include.

Conversion Specifier	Description
%s	Match a string.
%d	Match an integer in base 10 format.
%g	Match a double-precision floating-point value.

You can also specify that fscanf skip a value by specifying an asterisk in a conversion specifier. For example, %*f means skip the floating-point value in the input data; %*d means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf

Despite all the similarities between the MATLAB and C versions of fscanf, there are some significant differences. For example, consider a file named moon.dat for which the contents are as follows:

```
3.654234533
2.71343142314
5.34134135678
```

The following code reads all three elements of this file into a matrix named MyData:

```
fid = fopen('moon.dat','r');
MyData = fscanf(fid,'%g');
status = fclose(fid);
```

Notice that this code does not use any loops. Instead, the fscanf function continues to read in text as long as the input format is compatible with the format specifier.

An optional size argument controls the number of matrix elements read. For example, if fid refers to an open file containing strings of integers, then this line reads 100 integer values into the column vector A:

```
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix A:

```
A = fscanf(fid,'%5d',[10 10]);
```

A related function, sscanf, takes its input from a string instead of a file. For example, this line returns a column vector containing 2 and its square root:

```
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```

Writing Formatted Text Files

The fprintf function converts data to character strings and outputs them to the screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; fprintf copies text directly.

Common conversion specifiers include

Conversion Specifier	Description
%e	Exponential notation
%f	Fixed-point notation
%g	Automatically select the shorter of %e and %f

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function:

```
x = 0:0.1:1;
y = [x; exp(x)];
```

The code below writes x and y into a newly created file named exptable.txt:

```
fid = fopen('exptable.txt','w');
fprintf(fid,'Exponential Function\n\n');
fprintf(fid,'%6.2f  %12.8f\n',y);
status = fclose(fid);
```

The first call to fprintf outputs a title, followed by two carriage returns. The second call to fprintf outputs the table of numbers. The format control string specifies the format for each line of the table:

- A fixed-point value of six characters with two decimal places
- Two spaces
- A fixed-point value of twelve characters with eight decimal places

fprintf converts the elements of array y in column order. The function uses the format string repeatedly until it converts all the array elements.

Now use fscanf to read the exponential data file:

```
fid = fopen('exptable.txt','r');
title = fgetl(fid);
[table,count] = fscanf(fid,'%f %f',[2 11]);
table = table';
status = fclose(fid);
```

The second line reads the file title. The third line reads the table of values, two floating-point values on each line, until it reaches end of file. count returns the number of values matched.

A function related to fprintf, sprintf, outputs its results to a string instead of a file or the screen. For example,

```
root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));
```

Closing a File

When you finish reading or writing, use fclose to close the file. For example, this line closes the file associated with file identifier fid:

```
status = fclose(fid);
```

This line closes all open files:

```
status = fclose('all');
```

Both forms return `0` if the file or files were successfully closed or `-1` if the attempt was unsuccessful.

MATLAB automatically closes all open files when you exit from MATLAB. It is still good practice, however, to close a file explicitly with `fclose` when you are finished using it. Not doing so can unnecessarily drain system resources.

Note Closing a file does not clear the file identifier variable `fid`. However, subsequent attempts to access a file through this file identifier variable will not work.

Exchanging Files over the Internet

In this section...

"Overview" on page 6-117
"Downloading Web Content and Files" on page 6-117
"Creating and Decompressing Zip Archives" on page 6-119
"Sending E-Mail" on page 6-120
"Performing FTP File Operations" on page 6-122

Overview

MATLAB provides functions for exchanging files over the Internet. You can exchange files using common protocols, such as File Transfer Protocol (FTP), Simple Mail Transport Protocol (SMTP), and HyperText Transfer Protocol (HTTP). In addition, you can create zip archives to minimize the transmitted file size, and also save and work with Web pages.

Downloading Web Content and Files

MATLAB provides two functions for downloading Web pages and files using HTTP: `urlread` and `urlwrite`. With the `urlread` function, you can read and save the contents of a Web page to a string variable in the MATLAB workspace. With the `urlwrite` function, you can save a Web page's content to a file.

Because it creates a string variable in the workspace, the `urlread` function is useful for working with the contents of Web pages in MATLAB. The `urlwrite` function is useful for saving Web pages to a local directory.

Note When using `urlread`, remember that only the HTML in that specific Web page is retrieved. The hyperlink targets, images, and so on will not be retrieved.

If you need to pass parameters to a Web page, the `urlread` and `urlwrite` functions let you use HTTP `post` and `get` methods. For more information, see the `urlread` and `urlwrite` reference pages.

Example — Using the urlread Function

The following procedure demonstrates how to retrieve the contents of the Web page containing the Recent File list at the MATLAB Central File Exchange, `http://www.mathworks.com/matlabcentral/fileexchange/index.jsp`. It assigns the results to a string variable, `recentFile`, and it uses the `strfind` function to search the retrieved content for a specific word:

1 Retrieve the Web page content with the `urlread` function:

```
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');
```

2 After retrieving the content, run the `strfind` function on the `recentFile` variable:

```
hits = strfind(recentFile,'Simulink');
```

If the file contains the word `Simulink`, MATLAB will store the matches in the `hits` variable.

While you can manually pass arguments using the URL, the `urlread` function also lets you pass parameters to a Web page using standard HTTP methods, including `post` and `form`. Using the HTTP `get` method, which passes parameters in the URL, the following code queries Google for the word `Simulink`:

```
s =
urlread('http://www.google.com/search','get',{'q','Simulink'})
```

For more information, see the `urlread` reference page.

Example — Using the urlwrite Function

The following example builds on the procedure in the previous section. This example still uses `urlread` and checks for a specific word, but it also uses `urlwrite` to save the file if it contains any matches:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.

recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.

urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');
end;
```

MATLAB saves the Web page as `contains_simulink.html`.

Creating and Decompressing Zip Archives

Using the `zip` and `unzip` functions, you can compress and decompress files and directories. The `zip` function compresses files or directories into a zip archive. The `unzip` function decompresses zip archives.

Example — Using the zip Function

Again building on the example from previous sections, the following code creates a zip archive of the retrieved Web page:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
```

```
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved Web
page.
zip('simulink_matches.zip','contains_simulink.html');
end;
```

Sending E-Mail

To send an e-mail from within MATLAB, use the `sendmail` function. You can also attach files to an e-mail, which lets you mail files directly from MATLAB. To use `sendmail`, you must first set up your e-mail address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- E-mail address: This preference sets your e-mail address that will appear on the message. Here is an example of the syntax:

  ```
  setpref('Internet','E_mail','youraddress@yourserver.com');
  ```

- SMTP server: This preference sets your outgoing SMTP server address, which can be almost any e-mail server that supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP). Here is an example of the syntax:

  ```
  setpref('Internet', 'SMTP_Server', 'mail.server.network');
  ```

You should be able to find your outgoing SMTP server address in your e-mail account settings in your e-mail client application. You can also contact your system administrator for the information.

Note The sendmail function does not support e-mail servers that require authentication.

Once you have properly configured MATLAB, you can use the sendmail function. The sendmail function requires at least two arguments: the recipient's e-mail address and the e-mail subject:

```
sendmail('recepient@someserver.com', 'Hello From MATLAB!');
```

You can supply multiple e-mail addresses using a cell array of strings, such as:

```
sendmail({'recepient@someserver.com', ...
'recepient2@someserver.com'}, 'Hello From MATLAB!');
```

You can also specify a message body with the sendmail function, such as:

```
sendmail('recepient@someserver.com', 'Hello From MATLAB!', ...
'Thanks for using sendmail.');
```

In addition, you can also attach files to an e-mail using the sendmail function, such as:

```
sendmail('recepient@somesever.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', 'C:\yourFileSystem\message.txt');
```

You cannot attach a file without including a message. However, the message can be empty. You can also attach multiple files to an e-mail with the sendmail function, such as:

```
sendmail('recepient@somesever.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', ...
{'C:\yourFileSystem\message.txt',...
'C:\yourFileSystem\message2.txt'});
```

Example — Using the sendmail Function

The following example sends e-mail with the retrieved Web page archive attached if it contains any matches for the specified word:

```
% The urlread function loads the contents of the Web page into
the  % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved web
page.
zip('simulink_matches.zip','contains_simulink.html');

% The setpref function supplies your e-mail address and SMTP
% server address to MATLAB.
setpref('Internet','SMTP_Server','mail.server.network');
setpref('Internet', 'E_mail', 'youraddress@yourserver.com');

% The sendmail function sends an e-mail with the zip archive of
the
% retrieved Web page attached.
sendmail('youraddress@yourserver.com', 'New Simulink Files
Found', 'New Simulink files uploaded to MATLAB Central. See
attached zip archive.', 'simulink_matches.zip');
end;
```

Performing FTP File Operations

From within MATLAB, you can connect to an FTP server to perform remote file operations. The following procedure uses a public MathWorks FTP server

(ftp.mathworks.com). To perform any file operation on an FTP server, follow these steps:

1. Connect to the server using the ftp function.

   For example, you can create an FTP object for the public MathWorks FTP server with tmw=ftp('ftp.mathworks.com').

2. Perform the file operations using appropriate MATLAB FTP functions as methods acting on the server object.

   For example, you can display the file directories on the FTP server with dir(tmw).

3. When you finish working on the server, close the connection object using the close function.

   For example, you can disconnect from the FTP server with close(tmw).

Example — Retrieving a File from an FTP Server

In this example, you retrieve the file pub/pentium/Moler_1.txt, which is on the MathWorks FTP server. You can run this example; the FTP server and content are valid.

1. Connect to the MathWorks FTP server using ftp. This creates the server object tmw:

   ```
   tmw=ftp('ftp.mathworks.com');
   ```

2. List the contents of the server using the dir FTP function, which operates on the server object tmw:

   ```
   dir(tmw)
   ```

3. Change to the pub directory by using the FTP cd function. As with all FTP functions, you need to specify the server object you created using ftp as part of the syntax. In this case, this is tmw:

   ```
   cd(tmw,'pub');
   ```

The server object tmw represents the current directory on the FTP server, which now is pub.

4 Now when you run

```
dir(tmw)
```

you see the contents of pub, rather than the top level contents as displayed previously when you ran dir(tmw).

5 Use mget to retrieve any of the files from the current directory on the FTP server to the MATLAB current directory:

```
mget(tmw,'filename');
```

6 Close the FTP connection using close.

```
close(tmw);
```

Summary of FTP Functions

The following table lists the available FTP functions. For more information, refer to the applicable reference pages.

Function	Description
ascii	Set FTP transfer type to ASCII (convert new lines).
binary	Set FTP transfer type to binary (transfer verbatim, default).
cd (ftp)	Change current directory on FTP server.
delete (ftp)	Delete file on FTP server.
dir (ftp)	List contents of directory on FTP server.
close (ftp)	Close connection with FTP server.
ftp	Connect to FTP server, creating an FTP object.
mget	Download file from FTP site.
mkdir (ftp)	Create new directory on FTP server.
mput (ftp)	Upload file or directory to FTP server.

Function	Description
rename	Rename file on FTP server.
rmdir (ftp)	Remove directory on FTP server.

7

Working with Scientific Data Formats

This section describes how to import and export data in several standard scientific data formats. Topics covered include

Common Data Format (CDF) Files (p. 7-2)	Reading and writing data and metadata using the Common Data Format (CDF) file format.
Flexible Image Transport System (FITS) Files (p. 7-8)	Reading data and metadata using the Flexible Image Transport System (FITS) file format.
Hierarchical Data Format (HDF5) Files (p. 7-11)	Reading and writing data and metadata using the Hierarchical Data Format (HDF5) file format.
Hierarchical Data Format (HDF4) Files (p. 7-36)	Reading and writing data and metadata using the Hierarchical Data Format (HDF4) file format.

Common Data Format (CDF) Files

In this section...

"Getting Information About CDF Files" on page 7-2
"Importing Data from a CDF File" on page 7-3
"Exporting Data to a CDF File" on page 7-6

Getting Information About CDF Files

To get information about the contents of a Common Data Format (CDF) file, use the `cdfinfo` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). The `cdfinfo` function returns a structure containing general information about the file and detailed information about the variables and attributes in the file. For more information about this format, see the CDF Web site.

The following example returns information about the sample CDF file included with MATLAB. To determine the variables contained in the file, view the `Variables` field. This field contains a cell array that lists all the variables in the file with information that describes the variable, such as name, size, and data type. For an example, see "Importing Data from a CDF File" on page 7-3.

Note Because `cdfinfo` creates temporary files, make sure that your current working directory is writable before attempting to use the function.

```
info = cdfinfo('example.cdf')

info =

              Filename: 'example.cdf'
           FileModDate: '09-Mar-2001 16:45:22'
              FileSize: 1240
                Format: 'CDF'
```

```
        FormatVersion: '2.7.0'
         FileSettings: [1x1 struct]
             Subfiles: {}
            Variables: {5x6 cell}
     GlobalAttributes: [1x1 struct]
   VariableAttributes: [1x1 struct]
```

Importing Data from a CDF File

To import data into the MATLAB workspace from a Common Data Format (CDF) file, use the `cdfread` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can import all the data in the file, specific variables, specific records, or subsets of the data in a specific variable. The following examples illustrate some of these capabilities.

1 To get information about the contents of a CDF file, such as the names of variables in the CDF file, use the `cdfinfo` function. In this example, the `Variables` field indicates that the file contains five variables. The first variable, `Time`, is made up of 24 records containing CDF epoch data. The next two variables, `Longitude` and `Latitude`, have only one associated record containing `int8` data. For details about how to interpret the data returned in the `Variables` field, see `cdfinfo`.

```
info = cdfinfo('example.cdf');

vars = info.Variables

vars =

  Columns 1 through 5

    'Time'                [1x2 double]    [24]    'epoch'     'T/'
    'Longitude'           [1x2 double]    [ 1]    'int8'      'F/FT'
    'Latitude'            [1x2 double]    [ 1]    'int8'      'F/TF'
    'Data'                [1x3 double]    [ 1]    'double'    'T/TTT'
    'multidimensional     [1x4 double]    [ 1]    'uint8'     'T/TTTT'

  Column 6
```

```
'Full'
'Full'
'Full'
'Full'
'Full'
```

2 To read all of the data in the CDF file, use the `cdfread` function. The function returns the data in a 24-by-5 cell array. The five columns of data correspond to the five variables; the 24 rows correspond to the 24 records associated with the Time variable and padding elements for the rows associated with the other variables. The padding value used is specified in the CDF file.

```
data = cdfread('example.cdf');

whos data
  Name       Size             Bytes  Class    Attributes

  data      24x5              14784  cell
```

3 To read the data associated with a particular variable, use the `'Variable'` parameter, specifying a cell array of variable names as the value of this parameter. Variable names are case sensitive. For example, the following code reads the Longitude and Latitude variables from the file. The return value `data` is a 24-by-2 cell array, where each cell contains `int8` data.

```
var_time = cdfread('example.cdf','Variable',{'Longitude','Latitude'});

whos var_time
  Name           Size            Bytes  Class    Attributes

  var_time      24x1              4608  cell
```

Speeding Up Read Operations

The `cdfread` function offers two ways to speed up read operations when working with large data sets:

- Reducing the number of elements in the returned cell array

- Returning CDF epoch values as MATLAB serial date numbers rather than as MATLAB cdfepoch objects

To reduce the number of elements in the returned cell array, specify the 'CombineRecords' parameter. By default, cdfread creates a cell array with a separate element for every variable and every record in each variable, padding the records dimension to create a rectangular cell array. For example, reading all the data from the example file produces an output cell array, 24-by-5, where the columns represent variables and the rows represent the records for each variable. When you set the 'CombineRecords' parameter to true, cdfread creates a separate element for each variable but saves time by putting all the records associated with a variable in a single cell array element. Thus, reading the data from the example file with 'CombineRecords' set to true produces a 1-by-5 cell array, as shown below.

```
data_combined = cdfread('example.cdf','CombineRecords',true);

whos
  Name                Size            Bytes  Class    Attributes

  data               24x5             14784  cell
  data_combined       1x5              2364  cell
```

When combining records, note that the dimensions of the data in the cell change. For example, if a variable has 20 records, each of which is a scalar value, the data in the cell array for the combined element contains a 20-by-1 vector of values. If each record is a 3-by-4 array, the cell array element contains a 20-by-3-by-4 array. For combined data, cdfread adds a dimension to the data, the first dimension, that is the index into the records.

Another way to speed up read operations is to read CDF epoch values as MATLAB serial date numbers. By default, cdfread creates a MATLAB cdfepoch object for each CDF epoch value in the file. If you specify the 'ConvertEpochToDatenum' parameter, setting it to true, cdfread returns CDF epoch values as MATLAB serial date numbers. For more information about working with MATLAB cdfepoch objects, see “Representing CDF Time Values” on page 7-6.

```
data_datenums = cdfread('example.cdf','ConvertEpochToDatenum',true);
```

```
whos
  Name                Size            Bytes  Class    Attributes

  data               24x5             14784  cell
  data_combined       1x5              2364  cell
  var_time           24x1              4608  cell
```

Representing CDF Time Values

CDF represents time differently than MATLAB. CDF represents date and time as the number of milliseconds since 1-Jan-0000. This is called an *epoch* in CDF terminology. MATLAB represents date and time as a serial date number, which is the number of days since 0-Jan-0000. To represent CDF dates, MATLAB uses an object called a CDF epoch object. To access the time information in a CDF object, use the object's `todatenum` method.

For example, this code extracts the date information from a CDF epoch object:

1. Extract the date information from the CDF epoch object returned in the cell array `data` (see "Importing Data from a CDF File" on page 7-3). Use the `todatenum` method of the CDF epoch object to get the date information, which is returned as a MATLAB serial date number.

   ```
   m_date = todatenum(data{1});
   ```

2. View the MATLAB serial date number as a string.

   ```
   datestr(m_date)
   ans =

   01-Jan-2001
   ```

Exporting Data to a CDF File

To export data from the MATLAB workspace to a Common Data Format (CDF) file, use the `cdfwrite` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can write variables and attributes to the file, specifying their names and associated values. See the `cdfwrite` reference page for more information.

This example shows how to write date information to a CDF file. Note how the example uses the CDF epoch object constructor, cdfepoch, to convert a MATLAB serial date number into a CDF epoch.

```
cdfwrite('myfile',{'Time_val',cdfepoch(now)});
```

You can convert a cdfepoch object back into a MATLAB serial date number with the todatenum function.

Flexible Image Transport System (FITS) Files

In this section...

"Getting Information About FITS Files" on page 7-8

"Importing Data from a FITS File" on page 7-9

Getting Information About FITS Files

To get information about the contents of a Flexible Image Transport System (FITS) file, use the `fitsinfo` function. The FITS file format is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). For more information about the FITS standard, go to the official FITS Web site, `http://fits.gsfc.nasa.gov/`.

A data file in FITS format can contain multiple components, each marked by an ASCII text header followed by binary data. The first component in a FITS file is known as the *primary*, which can be followed by any number of other components, called *extensions,* in FITS terminology. The `fitsinfo` function returns a structure containing the information about the file and detailed information about the data in the file. This example returns information about a sample FITS file included with MATLAB. The structure returned contains fields for the primary component, `PrimaryData`, and all the extensions in the file, such as the `BinaryTable`, `Image`, and `AsciiTable` extensions.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

Importing Data from a FITS File

To import data into the MATLAB workspace from a Flexible Image Transport System (FITS) file, use the `fitsread` function. The FITS file format is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and two-dimensional tables containing rows and columns of data. Using this function, you can import the data in the `PrimaryData` section of the file or you can import the data in any of the extensions in the file, such as the `Image` extension. This example illustrates how to use the `fitsread` function to read data from a FITS file:

1 Determine which extensions the FITS file contains, using the `fitsinfo` function.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

The `info` structure shows that the file contains several extensions including the `BinaryTable`, `AsciiTable`, and `Image` extensions.

2 Read data from the file.

To read the `PrimaryData` in the file, specify the filename as the only argument:

```
pdata = fitsread('tst0012.fits');
```

To read any of the extensions in the file, you must specify the name of the extension as an optional parameter. This example reads the `BinaryTable` extension from the FITS file:

```
bindata = fitsread('tst0012.fits','bintable');
```

Note To read the `BinaryTable` extension using `fitsread`, you must specify the parameter `'bintable'`. Similarly, to read the `AsciiTable` extension, you must specify the parameter `'table'`. See the `fitsread` reference page for more information.

Hierarchical Data Format (HDF5) Files

In this section...

"Using the MATLAB High-Level HDF5 Functions" on page 7-11

"Using the MATLAB Low-Level HDF5 Functions" on page 7-26

Note For information about working with HDF4 data, which is a completely separate, incompatible format, see "Hierarchical Data Format (HDF4) Files" on page 7-36.

Using the MATLAB High-Level HDF5 Functions

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

The MATLAB high-level HDF5 functions provide an easy way to import data or metadata from an HDF5 file, or write data to an HDF5 file. The following sections provide more detail about using these functions.

- "Determining the Contents of an HDF5 File" on page 7-11
- "Importing Data from an HDF5 File" on page 7-15
- "Exporting Data to HDF5 Files" on page 7-16
- "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-18

Determining the Contents of an HDF5 File

HDF5 files can contain data and metadata, called *attributes*. HDF5 files organize the data and metadata in a hierarchical structure similar to the hierarchical structure of a UNIX file system.

In an HDF5 file, the directories in the hierarchy are called *groups*. A group can contain other groups, data sets, attributes, links, and data types. A data set is a collection of data, such as a multidimensional numeric array or string. An attribute is any data that is associated with another entity, such as a data set. A link is similar to a UNIX file system symbolic link. Links are a way to reference data without having to make a copy of the data.

Data types are a description of the data in the data set or attribute. Data types tell how to interpret the data in the data set. For example, a file might contain a data type called "Reading" that is comprised of three elements: a longitude value, a latitude value, and a temperature value.

To explore the hierarchical organization of an HDF5 file, use the `hdf5info` function. For example, to find out what the sample HDF5 file, `example.h5`, contains, use this syntax:

```
fileinfo = hdf5info('example.h5');
```

`hdf5info` returns a structure that contains various information about the HDF5 file, including the name of the file and the version of the HDF5 library that MATLAB is using:

```
fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.6.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

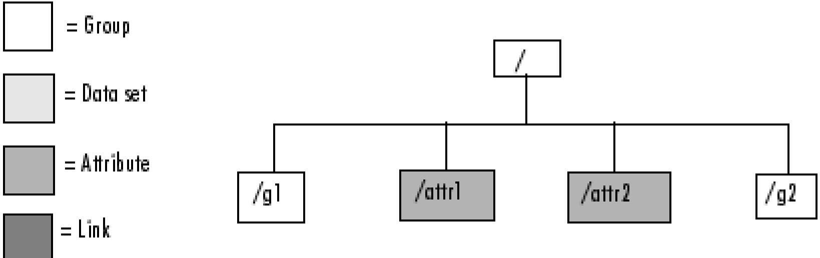
In the information returned by `hdf5info`, look at the `GroupHierarchy` field. This field is a structure that describes the top-level group in the file, called the *root* group. Using the UNIX convention, HDF5 names this top-level group `/` (forward slash), as shown in the `Name` field of the `GroupHierarchy` structure.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/'
```

```
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

By looking at the Groups and Attributes fields, you can see that the file contains two groups and two attributes. The Datasets, Datatypes, and Links fields are all empty, indicating that the root group does not contain any data sets, data types, or links.

The following figure illustrates the organization of the root group in the sample HDF5 file example.h5.

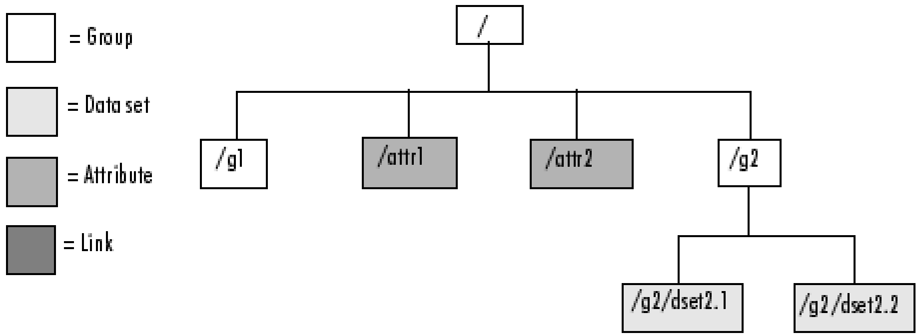


Organization of the Root Group of the Sample HDF5 File

To explore the contents of the sample HDF5 file further, examine one of the two structures in the Groups field of the GroupHierarchy structure. Each structure in this field represents a group contained in the root group. The following example shows the contents of the second structure in this field.

```
level2 = toplevel.Groups(2)

level2 =

      Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/g2'
        Groups: []
      Datasets: [1x2 struct]
     Datatypes: []
         Links: []
    Attributes: []
```

In the sample file, the group named /g2 contains two data sets. The following figure illustrates this part of the sample HDF5 file organization.

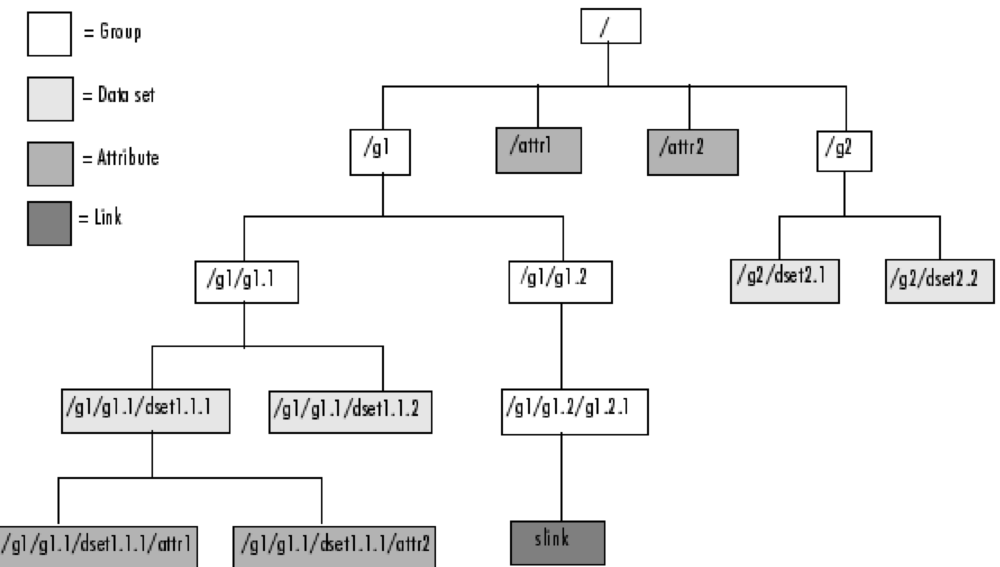


Organization of the Data Set /g2 in the Sample HDF5 File

To get information about a data set, look at either of the structures returned in the Datasets field. These structures provide information about the data set, such as its name, dimensions, and data type.

```
dataset1 = level2.Datasets(1)

dataset1 =
      Filename: 'L:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/g2/dset2.1'
          Rank: 1
      Datatype: [1x1 struct]
          Dims: 10
       MaxDims: 10
        Layout: 'contiguous'
    Attributes: []
         Links: []
     Chunksize: []
     Fillvalue: []
```

By examining the structures at each level of the hierarchy, you can traverse the entire file. The following figure describes the complete hierarchical organization of the sample file `example.h5`.



Hierarchical Structure of example.h5 HDF5 File

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `hdf5read` function. As arguments, you must specify the name of the HDF5 file and the name of the data set or attribute. Alternatively, you can specify just the field in the structure returned by `hdf5info` that contains the name of the data set or attribute; `hdf5read` can determine the file name from the `Filename` field in the structure. For more information about finding the name of a data set or attribute in an HDF5 file, see "Determining the Contents of an HDF5 File" on page 7-11.

To illustrate, this example reads the data set, `/g2/dset2.1` from the HDF5 sample file `example.h5`.

```
data = hdf5read('example.h5','/g2/dset2.1');
```

The return value contains the values in the data set, in this case a 1-by-10 vector of single-precision values:

```
data =

     1.0000
     1.1000
     1.2000
     1.3000
     1.4000
     1.5000
     1.6000
     1.7000
     1.8000
     1.9000
```

The hdf5read function maps HDF5 data types to appropriate MATLAB data types, whenever possible. If the HDF5 file contains data types that cannot be represented in MATLAB, hdf5write uses one of the predefined MATLAB HDF5 data type objects to represent the data.

For example, if an HDF5 data set contains four array elements, hdf5read can return the data as a 1-by-4 array of hdf5.h5array objects:

```
whos

Name    Size        Bytes     Class

data     1x4                hdf5.h5array

Grand total is 4 elements using 0 bytes
```

For more information about the MATLAB HDF5 data type objects, see "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-18.

Exporting Data to HDF5 Files

To write data or metadata from the MATLAB workspace to an HDF5 file, use the hdf5write function. As arguments, specify:

- Name of an existing HDF5 file, or the name you want to assign to a new file.

- Name of an existing data set or attribute, or the name you want to assign to a new data set or attribute. To learn how to determine the name of data sets in an existing HDF5 file, see "Determining the Contents of an HDF5 File" on page 7-11.
- Data or metadata you want to write to the file. `hdf5write` converts MATLAB data types to the appropriate HDF5 data type automatically. For nonatomic data types, you can also create HDF5 objects to represent the data.

This example creates a 5-by-5 array of `uint8` values and then writes the array to an HDF5 file. By default, `hdf5write` overwrites the file, if it already exists. The example specifies an `hdf5write` mode option to append data to existing file.

1. Create a MATLAB variable in the workspace. This example creates a 5-by-5 array of `uint8` values.

   ```
   testdata = uint8(magic(5))
   ```

2. Write the data to an HDF5 file. As arguments to `hdf5read`, the example specifies the name you want to assign to the HDF5 file, the name you want to assign to the data set, and the MATLAB variable.

   ```
   hdf5write('myfile.h5', '/dataset1', testdata)
   ```

   To add data to an existing file, you must use the `'writemode'` option, specifying the`'append'` value. The file must already exist and it cannot already contain a data set with the same name

   ```
   hdf5write('myfile.h5', '/dataset12', testdata,'writemode','append')
   ```

If you are writing simple data sets, such as scalars, strings, or a simple compound data types, you can just pass the data directly to `hdf5write`; this function automatically maps the MATLAB data types to appropriate HDF5 data types. However, if your data is a complex data set, you might need to use one of the predefined MATLAB HDF5 objects to pass the data to the `hdf5write` function. The HDF5 objects are designed for situations where the mapping between MATLAB and HDF5 types is ambiguous.

For example, when passed a cell array of strings, the `hdf5write` function writes a data set made up of strings, not a data set of arrays containing

strings. If that is not the mapping you intend, use HDF5 objects to specify the correct mapping. In addition, note that HDF5 makes a distinction between the size of a data set and the size of a data type. In MATLAB, data types are always scalar. In HDF5, data types can have a size; that is, types can be either scalar (like MATLAB) or m-by-n. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. For more information about the MATLAB HDF5 data type objects, see "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-18.

Mapping HDF5 Data Types to MATLAB Data Types

When the `hdf5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types to MATLAB data types, depending on whether the data in the data set is in an *atomic* data type or a nonatomic *composite* data type.

Mapping Atomic Data Types. Atomic data types describe commonly used binary formats for numbers (integers and floating point) and characters (ASCII). Because different computing architectures and programming languages support different number and character representations, the HDF5 library provides the platform-independent data types, which it then maps to an appropriate data type for each platform. For example, a computer may support 8-, 16-, 32-, and 64-bit signed integers, stored in memory in little endian byte order.

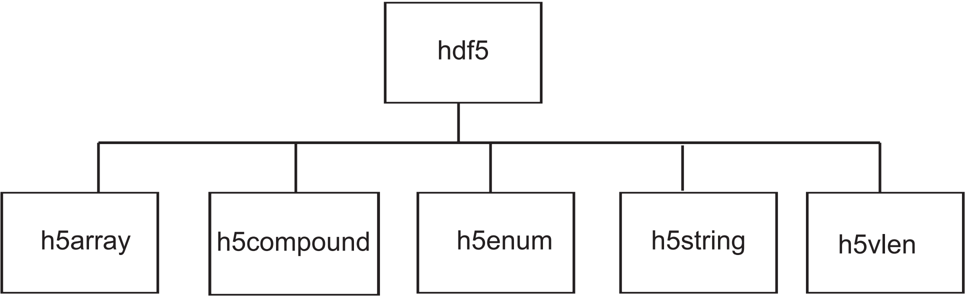
If the data in the data set is stored in one of the HDF5 atomic data types, `hdf5read` uses the equivalent MATLAB data type to represent the data. Each data set contains a `Datatype` field that names the data type. For example, the data set `/g2/dset2.2` in the sample HDF5 file includes atomic data and data type information.

```
dtype = dataset1.Datatype
dtype =

        Name: []
       Class: 'H5T_IEEE_F32BE'
    Elements: []
```

The H5T_IEEE_F32BE class name indicates the data is a 4-byte, big endian, IEEE floating-point data type. (See the HDF5 specification for more information about atomic data types.)

Mapping Composite Data Types. A composite data type is an aggregation of one or more atomic data types. Composite data types include structures, multidimensional arrays, and variable-length data types (one-dimensional arrays).

If the data in the data set is stored in one of the HDF5 nonatomic data types and the data cannot be represented in the workspace using a native MATLAB data type,hdf5read uses one of a set of classes MATLAB defines to represent HDF5 data types. The following figure illustrates the hdf5 class and its subclasses. For more information about a specific class, see the sections that follow. To learn more about the HDF5 data types in general, see the HDF Web page at http://www.hdfgroup.org.



For example, if an HDF5 file contains a data set made up of an enumerated data type which cannot be represented in MATLAB, hdf5read uses the HDF5 h5enum class to represent the data. An h5enum object has data members that store the enumerations (text strings), their corresponding values, and the enumerated data.

You might also need to use these HDF5 data type classes when using the hdf5write function to write data from the MATLAB workspace to an HDF5 file. By default, hdf5write can convert most MATLAB data to appropriate HDF5 data types. However, if this default data type mapping is not suitable, you can create HDF5 data types directly.

To access the data in the data set in the MATLAB workspace, you must access the `Data` field in the object.

This example converts a simple MATLAB vector into an `h5array` object and then displays the fields in the object:

```
vec = [ 1 2 3];

hhh = hdf5.h5array(vec);

hhh:

    Name: ''
    Data: [1 2 3]

hhh.Data

ans =

     1     2     3
```

MATLAB HDF5 h5array Data Class. The `h5array` data class associates a name with an array. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`arr = hdf5.h5array`	Creates an `h5array` object.
`arr = hdf5.h5array(data)`	Creates an `h5array` object, where `data` specifies the value of the `Data` member. `data` can be numeric, a cell array, or an HDF5 data type.

Methods	Description
`setData(arr, data)`	Sets the value of the `Data` member, where `arr` is an `h5array` object and `data` can be numeric, a cell array, or an HDF5 data type.
`setName(arr, name)`	Sets the value of the `Name` member, where `arr` is an `h5array` object and `name` is a string or cell array.

MATLAB HDF5 h5compound Data Class. The `h5compound` data class associates a name with a structure. You can define the field names in the structure and their values. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`MemberNames`	Text strings specifying name of the object

Methods	Description
`C = hdf5.h5compound`	Creates an `h5compound` object.
`C = hdf5.h5compound(n1,n2,...)`	Creates an `h5compound` object, where `n1`, `n2` and so on are text strings that specify field names. The constructor creates a corresponding data field for every member name.
`addMember(C, mName)`	Creates a new field in the object `C`. `mName` specifies the name of the field.
`setMember(C, mName, mData)`	Sets the value of the `Data` element associated with the field specified by `mName`, where `C` is an `h5compound` object and `mData` can be numeric or an HDF5 data type.

Methods	Description
`setMemberNames(C, n1, n2,...)`	Specifies the names of fields in the structure, where `C` is an `h5compound` object and `n1`,`n2`, and so on are text strings that specify field names. The method creates a corresponding data field for every name specified.
`setName(C, name)`	Sets the value of the `Name` member, where `C` is an `h5compound` object and `name` is a string or cell array.

MATLAB HDF5 h5enum Data Class. The `h5enum` data class defines an enumerated type. You can specify the enumerations (text strings) and the values they represent. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`EnumNames`	Text string specifying the enumerations, that is, the text strings that represent values
`EnumValues`	Values associated with enumerations

Methods	Description
`E = hdf5.h5enum`	Creates an `h5enum` object.
`E = hdf5.h5enum(eNames, eVals)`	Creates an `h5enum` object, where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.
`defineEnum(E, eNames, eVals)`	Defines the set of enumerations with the integer values they represent where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.

Methods	Description
enumdata = getString(E)	Returns a cell array containing the names of the enumerations, where E is an h5enum object.
setData(E, eData)	Sets the value of the object's Data member, where E is an h5enum object and eData is a vector of integers.
setEnumNames(E, eNames)	Specifies the enumerations, where E is an h5enum object and eNames is a cell array of strings.
setEnumValues(E, eVals)	Specifies the value associated with each enumeration, where E is an h5enum object and eVals is a vector of integers.
setName(E, name)	Sets the value of the object's Name member, where E is an h5enum object and name is a string or cell array.

This example uses an HDF5 enumeration object.

1 Create an HDF5 enumerated object.

```
enum_obj = hdf5.h5enum;
```

2 Define the enumerated values and their corresponding names.

```
enum_obj.defineEnum({'RED' 'GREEN' 'BLUE'}, uint8([1 2 3]));
```

enum_obj now contains the definition of the enumeration that associates the names RED, GREEN, and BLUE with the numbers 1, 2, and 3.

3 Add enumerated data to the object.

```
enum_obj.setData(uint8([2 1 3 3 2 3 2 1]));
```

In the HDF5 file, these numeric values map to the enumerated values GREEN, RED, BLUE, BLUE, GREEN, etc.

4 Write the enumerated data to a data set named objects in an HDF5 file.

```
hdf5write('myfile3.h5', '/g1/objects', enum_obj);
```

5 Read the enumerated data set from the file.

```
ddd = hdf5read('myfile3.h5','/g1/objects')

hdf5.h5enum:

          Name: ''
          Data: [2 1 3 3 2 3 2 1]
     EnumNames: {'RED'  'GREEN'  'BLUE'}
    EnumValues: [1 2 3]
```

MATLAB HDF5 h5string Data Class. The `h5string` data class associates a name with a text string and provides optional padding behavior. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`Length`	Scalar defining length of string

`Padding`	Type of padding to use:
	`'spacepad'`
	`'nullterm'`
	`'nullpad'`

Methods	Description
`str = hdf5.h5string`	Creates an `h5string` object.
`str = hdf5.h5string(data)`	Creates an `h5string` object, where `data` is a text string.
`str = hdf5.h5string(data, padtype)`	Creates an `h5string`object, where `data` is a text string and `padtype` specifies the type of padding to use.
`setData(str, data)`	Sets the value of the object's `Data` member, where `str` is an `h5string` object and`data` is a text string.

Methods	Description
setLength(str, lenVal)	Sets the value of the object's Length member, where str is an h5string object and lenVal is a scalar.
setName(str, name)	Sets the value of the object's Name member, where str is an h5string object and name is a string or cell array.
setPadding(str, padType)	Specifies the value of the object's Padding member, where str is an h5string object and padType is a text string specifying one of the supported pad types.

The following example creates an HDF5 string object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

The following example creates an HDF5 h5vlen object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

MATLAB HDF5 h5vlen Data Class. The h5vlen data class associates a name with an array. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`V = hdf5.h5vlen`	Creates an `h5vlen` object.
`V = hdf5.h5vlen(data)`	Creates an `h5vlen` object, where `data` specifies the value of the `Data` member. `data` can be numeric, a cell array, or an HDF5 data type.
`setData(V, data)`	Sets the value of the object's `Data` member, where `V` is an `h5vlen` object and `data` can be a scalar, vector, text string, a cell array, or an HDF5 data type.
`setName(V, name)`	Sets the value of the object's`Name` member, where `V` is an `h5vlen` object and `name` is a string or cell array.

Using the MATLAB Low-Level HDF5 Functions

MATLAB provides direct access to the over 200 functions in the HDF5 library by creating MATLAB functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities.

The HDF5 library organizes the library functions into groups, called *interfaces*. For example, all the routines related to working with files, such as opening and closing, are in the H5F interface, where *F* stands for file. MATLAB organizes the low-level HDF5 functions into classes that correspond to each HDF5 interface. For example, the MATLAB functions that correspond to the HDF5 file interface (H5F) are in the @H5F class directory. For a complete list of the HDF5 interfaces and the corresponding MATLAB class directories, see `hdf5`.

The following sections provide more details about how to use the MATLAB HDF5 low-level functions. Topics covered include:

- "Mapping HDF5 Function Syntax to MATLAB Function Syntax" on page 7-27
- "Mapping Between HDF5 Data Types and MATLAB Data Types" on page 7-29
- "Example: Using the MATLAB HDF5 Low-level Functions" on page 7-31

Note This section does not attempt to describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, you must refer to the official HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

Mapping HDF5 Function Syntax to MATLAB Function Syntax

In most cases, the syntax of the MATLAB low-level HDF5 functions is identical to the syntax of the corresponding HDF5 library functions. For example, the following is the function signature of the `H5Fopen` function in the HDF5 library. In the HDF5 function signatures, `hid_t` and `herr_t` are HDF5 types that return numeric values that represent object identifiers or error status values.

```
hid_t H5Fopen(const char *name, unsigned flags, hid_t access_id ) /* C syntax */
```

In MATLAB, each function in an HDF5 interface is a method of a MATLAB class. To view the function signature for a function, specify the class directory name and then the function name, as in the following.

```
help @H5F/open
```

The following shows the signature of the corresponding MATLAB function. First note that, because it's a method of a class, you must use the dot notation to call the MATLAB function: `H5F.open`. This MATLAB function accepts the same three arguments as the HDF5 function: a text string for the name, an HDF5-defined constant for the flags argument, and an HDF5 property list ID. You use property lists to specify characteristics of many different HDF5 objects. In this case, it's a file access property list. Refer to the HDF5

documentation to see which constants can be used with a particular function and note that, in MATLAB, constants are passed as text strings.

```
file_id = H5F.open(name, flags, plist_id)
```

There are, however, some functions where the MATLAB function signature is different than the corresponding HDF5 library function. The following sections describe some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

- "Output Parameters Become Return Values" on page 7-28
- "String Length Parameters Unnecessary" on page 7-28
- "Use Empty Array to Specify NULL" on page 7-29
- "Specifying Multiple Constants" on page 7-29

Output Parameters Become Return Values. Some HDF5 library functions use function parameters to return data on the right-hand side (RHS) of the function signature, i.e. as input parameters. The corresponding MATLAB function, because MATLAB allows multiple return values, moves these output parameters to the left-hand side (LHS) of the function signature, i.e. as return values. To illustrate, look at the `H5Dread` function. This function returns data in the `buf` parameter.

```
herr_t H5Dread(hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
        hid_t file_space_id, hid_t xfer_plist_id, void * buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value. Note that the HDF5 error return is not used. In MATLAB, the nonzero or negative value `herr_t` return values become MATLAB errors. Use MATLAB `try-catch` statements to handle errors.

```
buf = H5D.read(dataset_id, mem_type_id, mem_space_id, file_space_id, plist_id)
```

String Length Parameters Unnecessary. The length parameter used by some HDF5 library functions to specify the length of string parameters are not necessary in the corresponding MATLAB function. For example, the `H5Aget_name` function in the HDF5 library includes a buffer as an output parameter and the size of the buffer as an input parameter.

```
ssize_t H5Aget_name(hid_t attr_id,size_t buf_size,char *buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter buf into a return value and drops the buf_size parameter:

```
attr_name = H5A.get_name(attr_id)
```

Use Empty Array to Specify NULL. The MATLAB functions use empty arrays ([]) where HDF5 library functions accept the value NULL. For example, the H5Dfill function in the HDF5 library accepts the value NULL in place of a specified fill value.

```
herr_t H5Dfill(const void *fill, hid_t fill_type_id, void *buf,
               hid_t buf_type_id, hid_t space_id ) /* C syntax */
```

When using the corresponding MATLAB function, you can specify an empty array ([]) instead of NULL.

Specifying Multiple Constants. Some functions in the HDF5 library require you to specify an array of constants. For example, in the H5Screate_simple function, if you want to specify that each dimension in the data space can be unlimited, you use the constant H5S_UNLIMITED for each dimension in maxdims. In MATLAB, because you pass constants as text strings, you must use a cell array to achieve the same result. The following code fragment provides an example of using a cell array to specify this constant for each dimension of this data space.

```
ds_id = H5S.create_simple(2,[3 4],{'H5S_UNLIMITED' 'H5S_UNLIMITED'});
```

Mapping Between HDF5 Data Types and MATLAB Data Types

When the HDF5 low-level functions read data from an HDF5 file or write data to an HDF5 file, the functions map HDF5 data types to MATLAB data types automatically.

For *atomic* data types, such as commonly used binary formats for numbers (integers and floating point) and characters (ASCII), the mapping is typically straightforward because MATLAB supports similar types. See the table Mapping Between HDF5 Atomic Data Types and MATLAB Data Types on page 7-30 for a list of these mappings.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of `uint8` values
Reference	Array of `uint8` values
String	MATLAB character arrays.

For *composite* data types, such as aggregations of one or more atomic data types into structures, multidimensional arrays, and variable-length data types (one-dimensional arrays), the mapping is sometimes ambiguous with reference to the HDF5 data type. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

If your data is a complex data set, you might need to create HDF5 data types directly to make sure you have the mapping you intend. See the table Mapping Between HDF5 Composite Data Types and MATLAB Data Types on page 7-31 for a list of the default mappings. You can specify the data type when you write data to the file using the `H5Dwrite` function. See the HDF5 data type interface documentation for more information.

Mapping Between HDF5 Composite Data Types and MATLAB Data Types

HDF5 Composite Data Type	MATLAB Data Type
Array	Extends the dimensionality of the data type which it contains. For example, an array of an array of integers in HDF5 would map onto a two dimensional array of integers in MATLAB.
Compound	MATLAB structure. Note: All structures representing HDF5 data in MATLAB are scalar.
Enumeration	Array of integers which each have an associated name
Variable Length	MATLAB 1-D cell arrays

Reporting Data Set Dimensions. The MATLAB low-level HDF5 functions report data set dimensions and the shape of data sets differently than the MATLAB high-level functions. For ease of use, the MATLAB high-level functions report data set dimensions consistent with MATLAB column-major indexing. To be consistent with the HDF5 library, and to support the possibility of nested data sets and complicated data types, the MATLAB low-level functions report array dimensions using the C row-major orientation.

Example: Using the MATLAB HDF5 Low-level Functions

This example shows how to use the MATLAB HDF5 low-level functions to write a data set to an HDF5 file and then read the data set from the file.

1. Create the MATLAB variable that you want to write to the HDF5 file. The examples creates a three-dimensional array of `uint8` data.

   ```
   testdata = uint8(ones(5,10,3));
   ```

2. Create the HDF5 file or open an existing file. The example creates a new HDF5 file, named `my_file.h5`, in the system temp directory.

   ```
   filename = fullfile(tempdir,'my_file.h5');

   fileID = H5F.create(filename,'H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT');
   ```

In HDF5, you use the H5Fcreate function to create a file. The example uses the MATLAB equivalent, H5F.create. As arguments, specify the name you want to assign to the file, the type of access you want to the file ('H5F_ACC_TRUNC' in the example), and optional additional characteristics specified by a file creation property list and a file access property list. This example uses default values for these property lists ('H5P_DEFAULT'). In the example, note how the C constants are passed to the MATLAB functions as strings. The function returns an ID to the HDF5 file.

3 Create the data set in the file to hold the MATLAB variable. In the HDF5 programming model, you must define the data type and dimensionality (data space) of the data set as separate entities.

   a Specify the data type used by the data set. In HDF5, you use the H5Tcopy function to create integer or floating-point data types. The example uses the corresponding MATLAB function, H5T.copy, to create a uint8 data type because the MATLAB data is uint8. The function returns a data type ID.

   ```
   datatypeID = H5T.copy('H5T_NATIVE_UINT8');
   ```

   b Specify the dimensions of the data set. In HDF5, you use the H5Screate_simple routine to create a data space. The example uses the corresponding MATLAB function, H5S.create_simple, to specify the dimensions. The function returns a data space ID.

   ```
   dims(1) = 5;
   dims(2) = 10;
   dims(3) = 3
   dataspaceID = H5S.create_simple(3, dims, []);
   ```

   c Create the data set. In HDF5, you use the H5Dcreate routine to create a data set. The example uses the corresponding MATLAB function, H5D.create, specifying the file ID, the name you want to assign to the data set, data type ID, the data space ID, and a data set creation property list ID as arguments. The example uses the defaults for the property lists. The function returns a data set ID.

   ```
   dsetname = 'my_dataset';
   datasetID = H5D.create(fileID,dsetname,datatypeID,dataspaceID,'H5P_DEFAULT');
   ```

Note To write a large data set, you must use the chunking capability of the HDF5 library. To do this, create a property list and use the `H5P.set_chunk` function to set the chunk size in the property list. In the following example, the dimensions of the data set are `dims = [2^16 2^16]` and the chunk size is 1024-by-1024. You then pass the property list as the last argument to the data set creation function, `H5D.create`, instead of using the `H5P_DEFAULT` value.

```
plistID = H5P.create('H5P_DATASET_CREATE'); % create property list

chunk_size = min([1024 1024], dims); % define chunk size
H5P.set_chunk(plistID, chunk_size); % set chunk size in property list

datasetID = H5D.create(fileID, dsetname, datatypeID, dataspaceID, plistID);
```

4 Write the data to the data set. In HDF5, you use the `H5Dwrite` routine to write data to a data set. The example uses the corresponding MATLAB function, `H5D.write`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, the transfer property list ID and the name of the MATLAB variable to be written to the data set.

   You can use the memory data type to specify the data type used to represent the data in the file. The example uses the constant `'H5ML_DEFAULT'` which lets the MATLAB function do an automatic mapping to HDF5 data types. The memory data space ID and the data set's data space ID specify to write subsets of the data set to the file. The example uses the constant `'H5S_ALL'` to write all the data to the file and uses the default property list.

   Note Because HDF5 stores data in row-major order and MATLAB accesses data in column-major order, you should permute your data before writing it to the file.

```
data_perm = permute(testdata,[3 2 1]);
```

```
H5D.write(datasetID,'H5ML_DEFAULT','H5S_ALL','H5S_ALL', ...
          'H5P_DEFAULT',data_perm);
```

5 Close the data set, data space, data type, and file objects. If used inside a MATLAB function, these identifiers are closed automatically when they go out of scope.

```
H5D.close(datasetID);
H5S.close(dataspaceID);
H5T.close(datatypeID);
H5F.close(fileID);
```

6 To read the data set you wrote to the file, you must open the file. In HDF5, you use the `H5Fopen` routine to open an HDF5 file, specifying the name of the file, the access mode, and a property list as arguments. The example uses the corresponding MATLAB function, `H5F.open`, opening the file for read-only access.

```
fileID = H5F.open(filename,'H5F_ACC_RDONLY','H5P_DEFAULT');
```

7 After opening the file, you must open the data set. In HDF5, you use the `H5Dopen` function to open a data set. The example uses the corresponding MATLAB function, `H5D.open`, specifying as arguments the file ID and the name of the data set, defined earlier in the example.

```
datasetID = H5D.open(fileID, dsetname);
```

8 After opening the data set, you can read the data into the MATLAB workspace. In HDF5, you use the `H5Dread` function to read an HDF5 file. The example uses the corresponding MATLAB function, `H5D.read`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, and the transfer property list ID.

```
returned_data = H5D.read(datasetID,'H5ML_DEFAULT',...
                         'H5S_ALL','H5S_ALL','H5P_DEFAULT');
```

Note that the data returned must be indexed in reverse order: HDF5 stores the data in row-major order; MATLAB accesses data in column-major order. To rearrange the data into column-major order, use the MATLAB `permute` function.

```
data = permute(returned_data,[3 2 1]);
```

You can compare the original MATLAB variable, `testdata`, with the variable just created, `data`, to see if they are the same.

Hierarchical Data Format (HDF4) Files

In this section...

"Using the HDF Import Tool" on page 7-36
"Using the HDF Import Tool Subsetting Options" on page 7-41
"Using the MATLAB HDF4 High-Level Functions" on page 7-53
"Using the HDF4 Low-Level Functions" on page 7-56

Note For information about importing HDF5 data, which is a separate, incompatible format, see "Hierarchical Data Format (HDF5) Files" on page 7-11.

Using the HDF Import Tool

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (`www.hdfgroup.org`).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (`www.hdfeos.org`).

The HDF Import Tool is a graphical user interface that you can use to navigate through HDF4 or HDF-EOS files and import data from them. Importing data using the HDF Import Tool involves these steps:

- "Step 1: Opening an HDF4 File in the HDF Import Tool" on page 7-37
- "Step 2: Selecting a Data Set in an HDF File" on page 7-38
- "Step 3: Specifying a Subset of the Data (Optional)" on page 7-39
- "Step 4: Importing Data and Metadata" on page 7-40

- "Step 5: Closing HDF Files and the HDF Import Tool" on page 7-41

The following sections provide more detail about each of these steps.

Step 1: Opening an HDF4 File in the HDF Import Tool

Open an HDF4 or HDF-EOS file in MATLAB using one of the following methods:

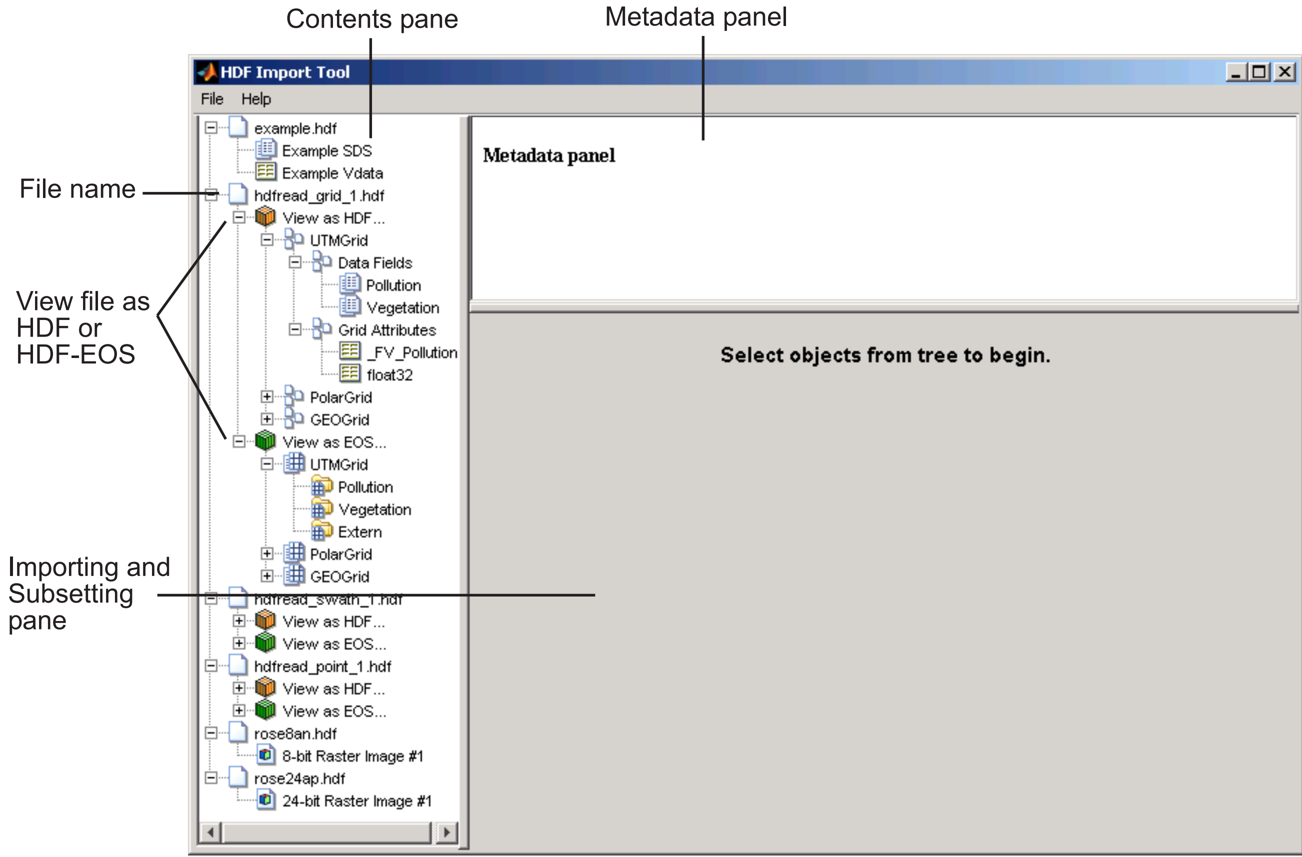
- Choose the **Import Data** option from the MATLAB **File** menu. If you select an HDF4 or HDF-EOS file, the MATLAB Import Wizard automatically starts the HDF Import Tool.
- Start the HDF Import Tool by entering the `hdftool` command at the MATLAB command line:

  ```
  hdftool
  ```

  This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.
- Open an HDF or HDF-EOS file by specifying the file name with the `hdftool` command on the MATLAB command line:

  ```
  hdftool('example.hdf')
  ```

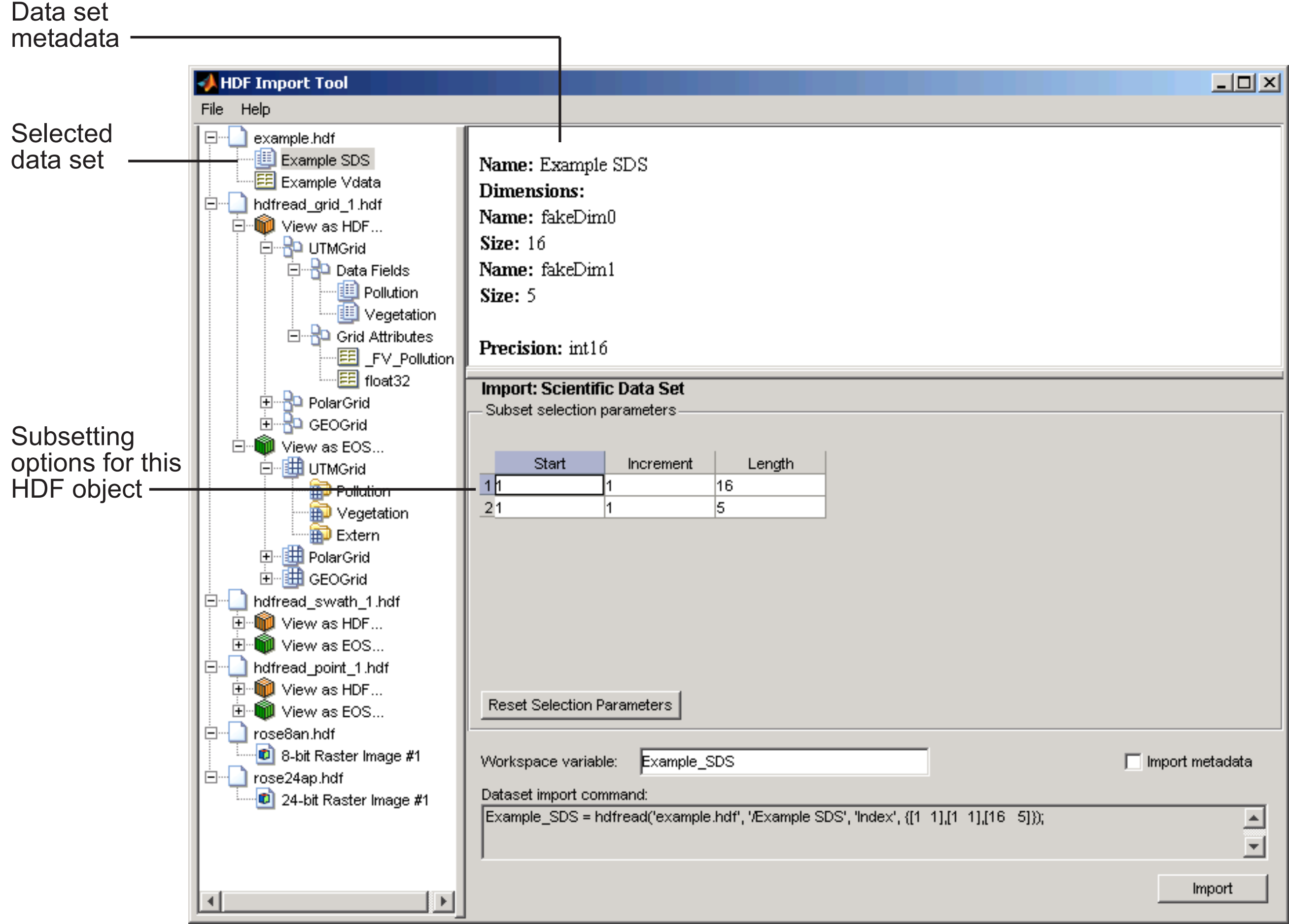
Viewing a File in the HDF Import Tool. When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon in the contents pane indicates the view, as illustrated in the following figure. Note that these are just two views of the same data.



Step 2: Selecting a Data Set in an HDF File

To import a data set, you must first select the data set in the contents pane of the HDF Import Tool. Use the Contents pane to view the contents of the file and navigate to the data set you want to import.

For example, the following figure shows the data set `Example SDS` in the HDF file selected. Once you select a data set, the Metadata panel displays information about the data set and the importing and subsetting pane displays subsetting options available for this type of HDF object.



Step 3: Specifying a Subset of the Data (Optional)

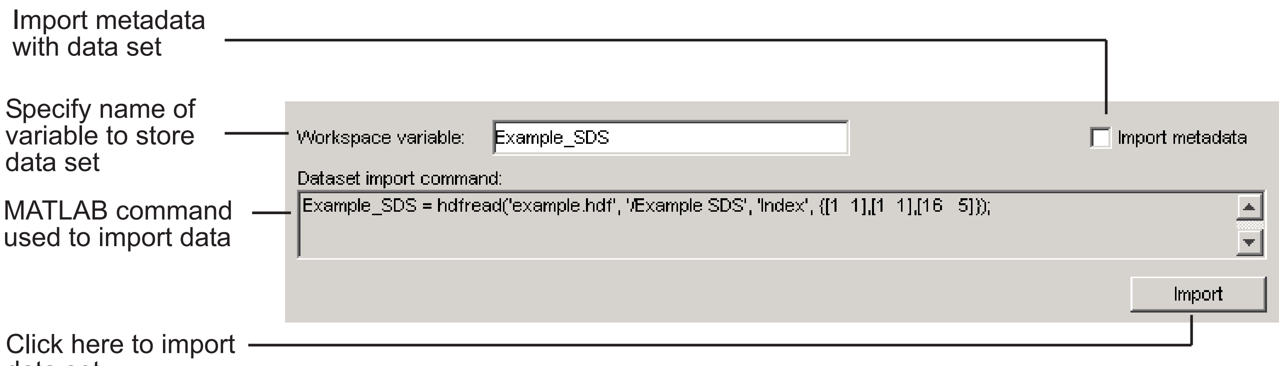
When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see "Using the HDF Import Tool Subsetting Options" on page 7-41.

Step 4: Importing Data and Metadata

To import the data set you have selected, click the **Import** button, bottom right corner of the Importing and Subsetting pane. Using the Importing and Subsetting pane, you can

- Specify the name of the workspace variable — By default, the HDF Import Tool uses the name of the HDF4 data set as the name of the MATLAB workspace variable. In the following figure, the variable name is `Example_SDS`. To specify another name, enter text in the **Workspace Variable** text box.
- Specify whether to import metadata associated with the data set — To import any metadata that might be associated with the data set, select the **Import Metadata** check box. To store the metadata, the HDF Import Tool creates a second variable in the workspace with the same name with "`_info`" appended to it. For example, if you select this check box, the name of the metadata variable for the data set in the figure would be `Example_SDS_info`.
- Save the data set import command syntax — The **Dataset import command** text window displays the MATLAB command used to import the data set. This text is not editable, but you can copy and paste it into the MATLAB Command Window or a text editor for reuse.

The following figure shows how to specify these options in the HDF Import Tool.



Step 5: Closing HDF Files and the HDF Import Tool

To close a file, select the file in the contents pane and click **Close File** on the HDF Import Tool **File** menu.

To close all the files open in the HDF Import Tool, click **Close All Files** on the HDF Import Tool **File** menu.

To close the tool, click **Close HDFTool** in the HDF Import Tool **File** menu or click the **Close** button in the upper right corner of the tool.

If you used the `hdftool` syntax that returns a handle to the tool,

```
h = hdftool('example.hdf')
```

you can use the `close(h)` command to close the tool from the MATLAB command line.

Using the HDF Import Tool Subsetting Options

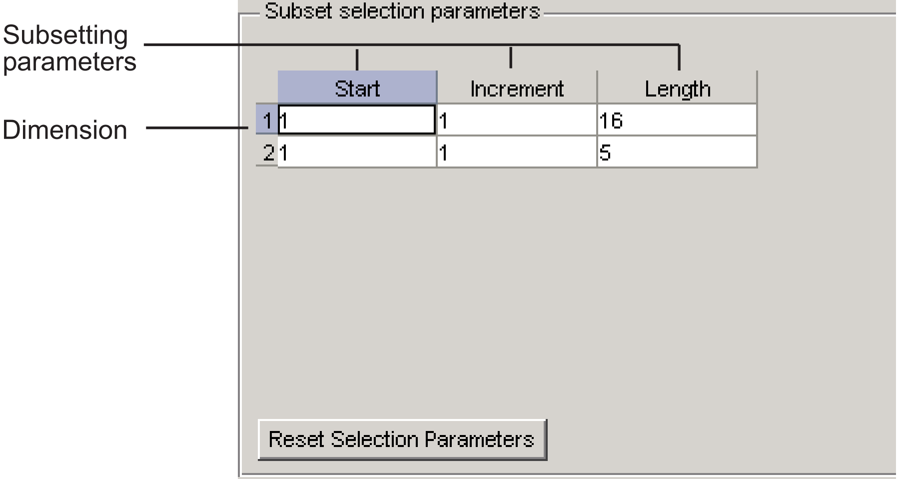
When you select a data set, the importing and subsetting pane displays the subsetting options available for that type of data set. The following sections provide information about these subsetting options for all supported data set types. For general information about the HDF Import tool, see "Using the HDF Import Tool" on page 7-36.

- "HDF Scientific Data Sets (SD)" on page 7-42
- "HDF Vdata" on page 7-42
- "HDF-EOS Grid Data" on page 7-44
- "HDF-EOS Point Data" on page 7-49
- "HDF-EOS Swath Data" on page 7-49
- "HDF Raster Image Data" on page 7-53

Note To use these data subsetting options effectively, you must understand the HDF and HDF-EOS data formats. Therefore, use this documentation in conjunction with the HDF documentation (`www.hdfgroup.org`) and the HDF-EOS documentation (`www.hdfeos.org`).

HDF Scientific Data Sets (SD)

The HDF scientific data set (SD) is a group of data structures used to store and describe multidimensional arrays of scientific data. Using the HDF Import Tool subsetting parameters, you can import a subset of an HDF scientific data set by specifying the location, range, and number of values to be read along each dimension.



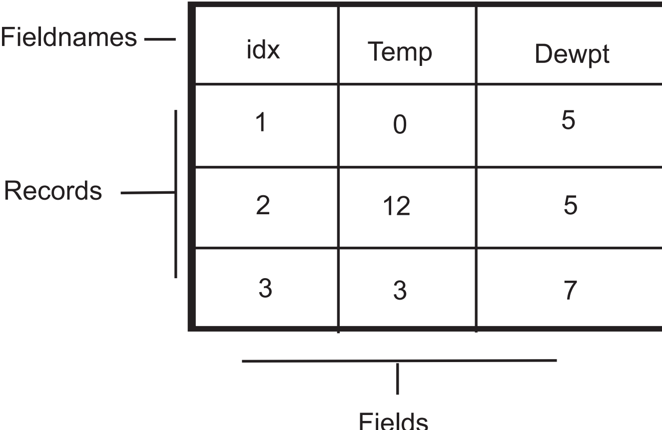
The subsetting parameters are:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

HDF Vdata

HDF Vdata data sets provide a framework for storing customized tables. A Vdata table consists of a collection of records whose values are stored in

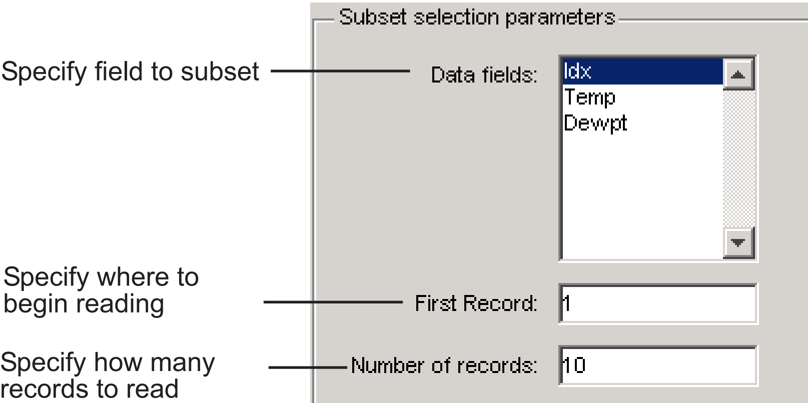
fixed-length fields. All records have the same structure and all values in each field have the same data type. Each field is identified by a name. The following figure illustrates a Vdata table.



You can import a subset of an HDF Vdata data set in the following ways:

- Specifying the name of the field that you want to import
- Specifying the range of records that you want to import

The following figure shows how you specify these subsetting parameters for Vdata.

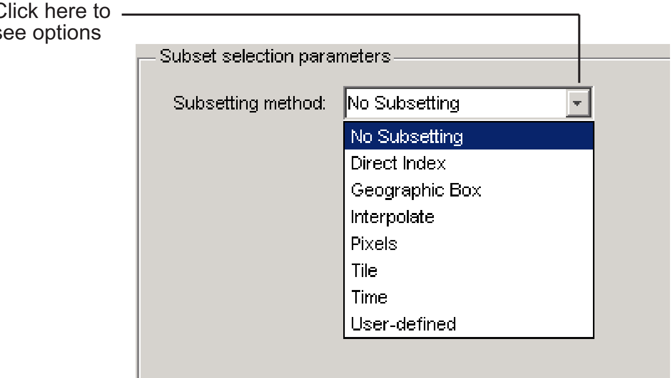


HDF-EOS Grid Data

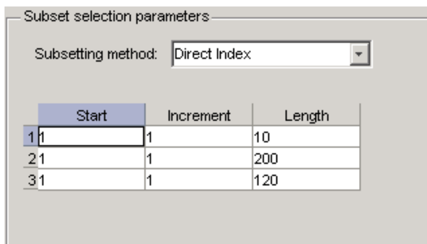
In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

- "Direct Index" on page 7-44
- "Geographic Box" on page 7-45
- "Interpolation" on page 7-46
- "Pixels" on page 7-47
- "Tile" on page 7-47
- "Time" on page 7-47
- "User-Defined" on page 7-48

To access these options, click the Subsetting method menu in the importing and subsetting pane.



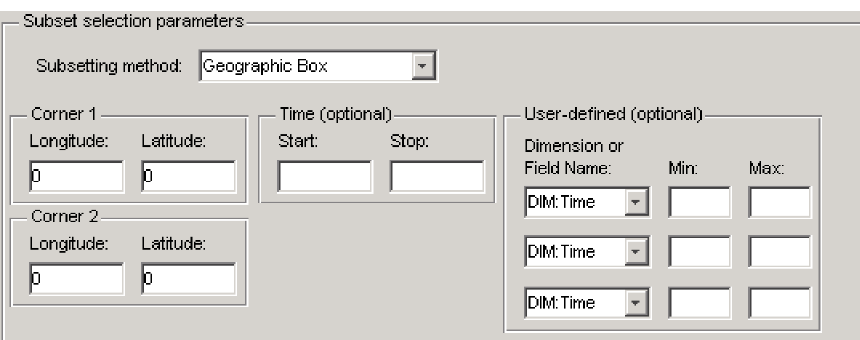
Direct Index. You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box. You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in. To define this rectangular area, you must specify two points, using longitude and latitude in decimal degrees. These points are two corners of the rectangular area. Typically, **Corner 1** is the upper-left corner of the box, and **Corner 2** is the lower-right corner of the box.

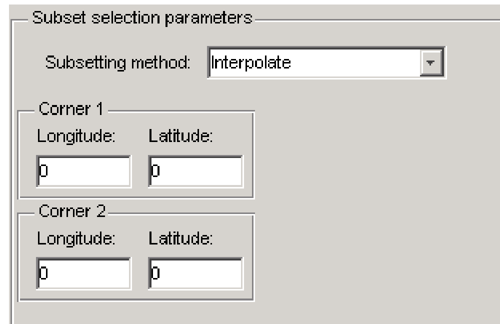


Optionally, you can further define the subset of data you are interested in by using Time parameters (see “Time” on page 7-47) or by specifying other User-Defined subsetting parameters (see “User-Defined” on page 7-48).

Interpolation. Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

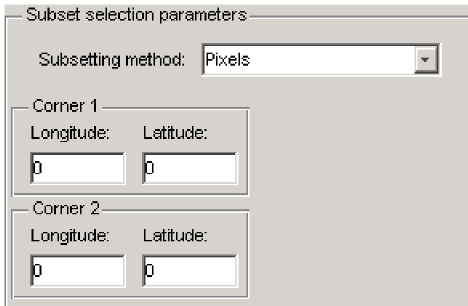
You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

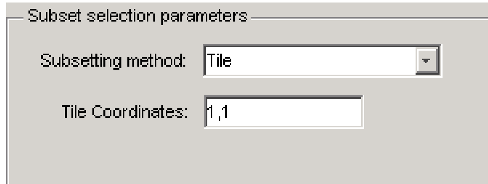


Pixels. You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid. You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

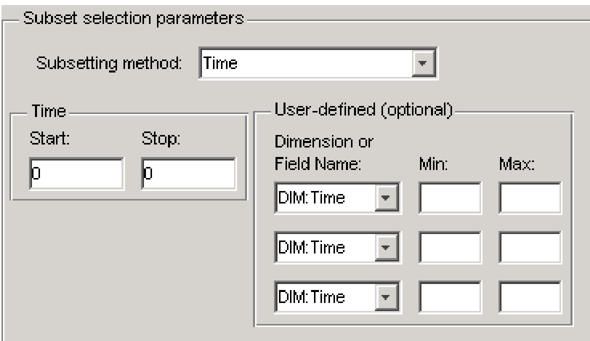
- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box



Tile. In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. If the HDF-EOS Grid data is stored as tiles, you can import a subset of the data by specifying the coordinates of the tile you are interested in. Tile coordinates are 1-based, with the upper-left corner of a two-dimensional data set identified as `1,1`. In a three-dimensional data set, this tile would be referenced as `1,1,1`.

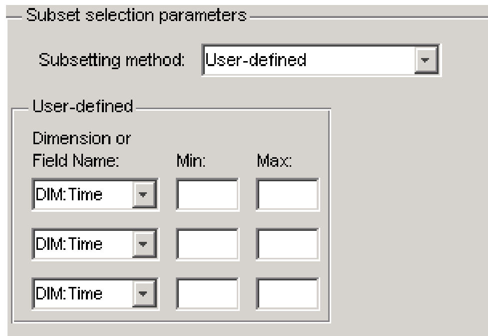


Time. You can import a subset of the Grid data set by specifying a time period. You must specify both the start time and the stop time (the endpoint of the time span). The units (hours, minutes, seconds) used to specify the time are defined by the data set.



Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters.

User-Defined. You can import a subset of the Grid data set by specifying user-defined subsetting parameters.

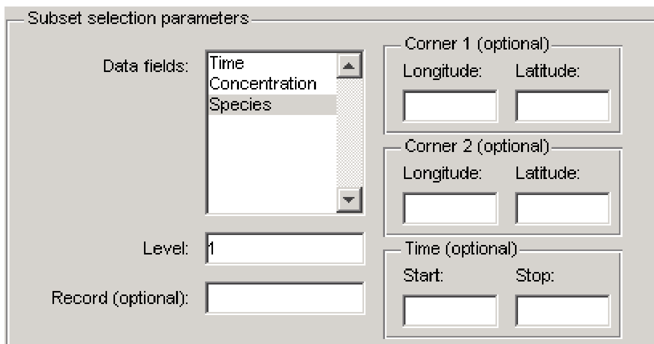


When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF-EOS Point Data

HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying field names and level. Optionally, you can refine the subsetting by specifying the range of records you want to import, by defining a rectangular area, or by specifying a time period. For information about specifying a rectangular area, see "Geographic Box" on page 7-45. For information about subsetting by time, see "Time" on page 7-47.



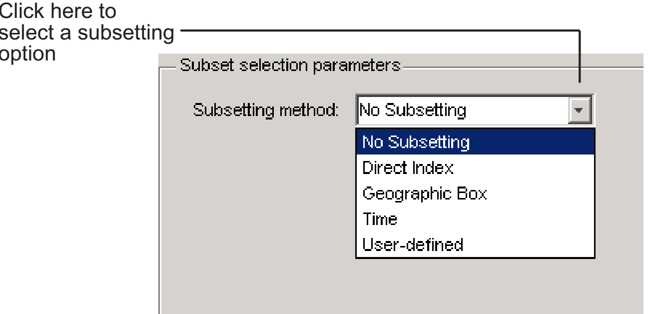
HDF-EOS Swath Data

HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

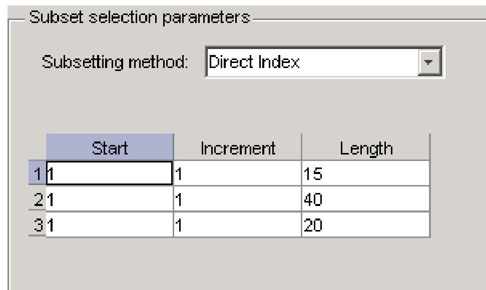
The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

- "Direct Index" on page 7-50
- "Geographic Box" on page 7-51
- "Time" on page 7-52
- "User-Defined" on page 7-52

To access these options, click the `Subsetting method` menu in the **Importing and Subsetting** pane.



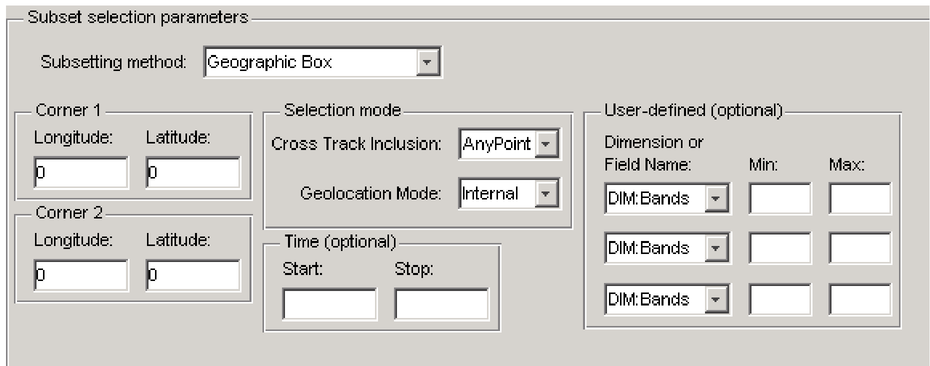
Direct Index. You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

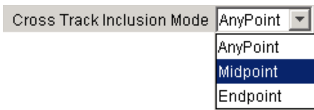
Geographic Box. You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in and by specifying the selection Mode.



You define the rectangular area by specifying two points that specify two corners of the box:

- **Corner 1** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

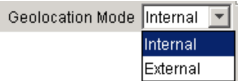


Select from these values:

- **AnyPoint** — Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath.

- **Endpoint** — All of the area defined by the box overlaps with the swath.

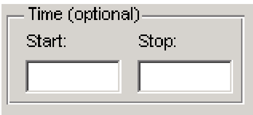
The **Geolocation Mode** value specifies whether geolocation fields and data must be in the same swath.



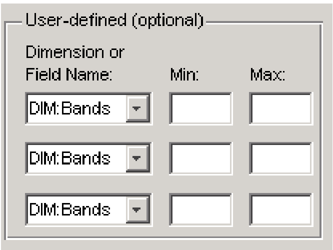
Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

Time. You can optionally also subset swath data by specifying a time period. The units used (hours, minutes, seconds) to specify the time are defined by the data set



User-Defined. You can optionally also subset a swath data set by specifying user-defined parameters.



When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF Raster Image Data

For 8-bit HDF raster image data, you can specify the colormap.

Using the MATLAB HDF4 High-Level Functions

To import data from an HDF or HDF-EOS file, you can use the MATLAB HDF4 high-level function `hdfread`. The `hdfread` function provides a programmatic way to import data from an HDF4 or HDF-EOS file that still hides many of the details that you need to know if you use the low-level HDF functions, described in "Using the HDF4 Low-Level Functions" on page 7-56. You can also import HDF4 data using an interactive GUI, described in "Using the HDF Import Tool" on page 7-36.

This section describes these high-level MATLAB HDF functions, including

- "Using hdfinfo to Get Information About an HDF4 File" on page 7-53
- "Using hdfread to Import Data from an HDF4 File" on page 7-54

To export data to an HDF4 file, you must use the MATLAB HDF4 low-level functions.

Using hdfinfo to Get Information About an HDF4 File

To get information about the contents of an HDF4 file, use the `hdfinfo` function. The `hdfinfo` function returns a structure that contains information about the file and the data in the file.

Note You can also use the HDF Import Tool to get information about the contents of an HDF4 file. See "Using the HDF Import Tool" on page 7-36 for more information.

This example returns information about a sample HDF4 file included with MATLAB:

```
info = hdfinfo('example.hdf')

info =

    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

To get information about the data sets stored in the file, look at the `SDS` field.

Using hdfread to Import Data from an HDF4 File

To use the`hdfread` function, you must specify the data set that you want to read. You can specify the filename and the data set name as arguments, or you can specify a structure returned by the `hdfinfo` function that contains this information. The following example shows both methods. For information about how to import a subset of the data in a data set, see "Reading a Subset of the Data in a Data Set" on page 7-56.

1 Determine the names of data sets in the HDF4 file, using the `hdfinfo` function.

   ```
   info = hdfinfo('example.hdf')

   info =

       Filename: 'example.hdf'
            SDS: [1x1 struct]
          Vdata: [1x1 struct]
   ```

   To determine the names and other information about the data sets in the file, look at the contents of the `SDS` field. The `Name` field in the SDS structure gives the name of the data set.

   ```
   dsets = info.SDS

   dsets =
   ```

```
     Filename: 'example.hdf'
         Type: 'Scientific Data Set'
         Name: 'Example SDS'
         Rank: 2
     DataType: 'int16'
   Attributes: []
         Dims: [2x1 struct]
        Label: {}
  Description: {}
        Index: 0
```

2 Read the data set from the HDF4 file, using the `hdfread` function. Specify the name of the data set as a parameter to the function. Note that the data set name is case sensitive. This example returns a 16-by-5 array:

```
dset = hdfread('example.hdf', 'Example SDS');

dset =

     3     4     5     6     7
     4     5     6     7     8
     5     6     7     8     9
     6     7     8     9    10
     7     8     9    10    11
     8     9    10    11    12
     9    10    11    12    13
    10    11    12    13    14
    11    12    13    14    15
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    17    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
```

Alternatively, you can specify the specific field in the structure returned by `hdfinfo` that contains this information. For example, to read a scientific data set, use the `SDS` field.

```
dset = hdfread(info.SDS);
```

Reading a Subset of the Data in a Data Set. To read a subset of a data set, you can use the optional `'index'` parameter. The value of the index parameter is a cell array of three vectors that specify the location in the data set to start reading, the skip interval (e.g., read every other data item), and the amount of data to read (e.g., the length along each dimension). In HDF4 terminology, these parameters are called the *start*, *stride*, and *edge* values.

For example, this code

- Starts reading data at the third row, third column (`[3 3]`).
- Reads every element in the array (`[]`).
- Reads 10 rows and 2 columns (`[10 2]`).

```
subset = hdfread('Example.hdf','Example SDS',...
                 'Index',{[3 3],[],[10 2 ]})

subset =

     7     8
     8     9
     9    10
    10    11
    11    12
    12    13
    13    14
    14    15
    15    16
    16    17
```

Using the HDF4 Low-Level Functions

This section describes how to use MATLAB functions to access the HDF4 Application Programming Interfaces (APIs). These APIs are libraries of C routines that you can use to import data from an HDF4 file or export data from the MATLAB workspace into an HDF4 file. To import or export data, you must use the functions in the HDF4 API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. To illustrate this concept, this section describes the programming

model of one particular HDF4 API: the HDF4 Scientific Data (SD) API. For a complete list of the HDF4 APIs supported by MATLAB and the functions you use to access each one, see the `hdf` reference page.

Note This section does not attempt to describe all HDF4 features and routines. To use the MATLAB HDF4 functions effectively, you must refer to the official NCSA documentation at the HDF Web site (`www.hdfgroup.org`).

Topics covered include

- "Understanding the HDF4 to MATLAB Syntax Mapping" on page 7-57
- "Example: Importing Data Using the HDF4 SD API Functions" on page 7-58
- "Example: Exporting Data Using the HDF4 SD API Functions" on page 7-64
- "Using the MATLAB HDF4 Utility API" on page 7-71

Understanding the HDF4 to MATLAB Syntax Mapping

Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (`SDopen`), close (`SDend`), and read data (`SDreaddata`).

Instead of supporting each routine in the HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in a particular HDF4 API. For example, the HDF Scientific Data (SD) API includes the C routine `SDend` to close an HDF4 file:

```
status = SDend(sd_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the SD API, `hdfsd`. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfsd('end',sd_id); % MATLAB code
```

Handling HDF4 Routines with Output Arguments. Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the `SDfileinfo` routine returns data about an HDF4 file in two output arguments, `ndatasets` and `nglobal_atts`. Here is the C code:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Example: Importing Data Using the HDF4 SD API Functions

To illustrate using HDF4 API routines in MATLAB, the following sections provide a step-by-step example of how to import HDF4 Scientific Data (SD) into the MATLAB workspace.

- “Step 1: Opening the HDF4 File” on page 7-59
- “Step 2: Retrieving Information About the HDF4 File” on page 7-59
- “Step 3: Retrieving Attributes from an HDF4 File (Optional)” on page 7-60
- “Step 4: Selecting the Data Sets to Import” on page 7-61
- “Step 5: Getting Information About a Data Set” on page 7-61
- “Step 6: Reading Data from the HDF4 File” on page 7-62
- “Step 7: Closing the HDF4 Data Set” on page 7-63
- “Step 8: Closing the HDF4 File” on page 7-64

Note The following sections, when referring to specific routines in the HDF4 SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

Step 1: Opening the HDF4 File. To import an HDF4 SD data set, you must first open the file using the SD API routine `SDstart`. (In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.) In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `start` in this case.
- Name of the file you want to open.
- Mode in which you want to open it. The following table lists the file access modes supported by the `SDstart` routine. In MATLAB, you specify these modes as text strings. You can specify the full HDF4 mode name or one of the abbreviated forms listed in the table.

HDF4 File Creation Mode	HDF4 Mode Name	MATLAB String
Create a new file	`'DFACC_CREATE'`	`'create'`
Read access	`'DFACC_RDONLY'`	`'read'` or `'rdonly'`
Read and write access	`'DFACC_RDWR'`	`'rdwr'` or `'write'`

For example, this code opens the file `mydata.hdf` for read access:

```
sd_id = hdfsd('start','mydata.hdf','read');
```

If `SDstart` can find and open the file specified, it returns an HDF4 SD file identifier, named `sd_id` in the example. Otherwise, it returns `-1`.

Step 2: Retrieving Information About the HDF4 File. To get information about an HDF4 file, you must use the SD API routine `SDfileinfo`. This function returns the number of data sets in the file and the number of global attributes in the file, if any. (For more information about global attributes, see “Example: Exporting Data Using the HDF4 SD API Functions” on page 7-64.) In MATLAB, you use the `hdfsd` function, specifying the following arguments:

- Name of the SD API routine, `fileinfo` in this case
- SD file identifier, `sd_id`, returned by `SDstart`

In this example, the HDF4 file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)

ndatasets =
    3

nglobal_atts =
    1

status =
    0
```

Step 3: Retrieving Attributes from an HDF4 File (Optional). HDF4 files can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an entire HDF4 file are called *global* attributes. Attributes associated with a data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information, see "Step 4: Writing Metadata to an HDF4 File" on page 7-69.)

To retrieve attributes from an HDF4 file, use the HDF4 API routine `SDreadattr`. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `readattr` in this case.
- File identifier (`sd_id`) returned by `SDstart`, for global attributes, or the data set identifier for local attributes. (See "Step 4: Selecting the Data Sets to Import" on page 7-61 to learn how to get a data set identifier.)
- Index identifying the attribute you want to view. HDF4 uses zero-based indexing. If you know the name of an attribute but not its index, use the `SDfindattr` routine to determine the index value associated with the attribute.

For example, this code returns the contents of the first global attribute, which is the character string `my global attribute`:

```
attr_idx = 0;
[attr, status] = hdfsd('readattr', sd_id, attr_idx);

attr =
    my global attribute
```

Step 4: Selecting the Data Sets to Import. To select a data set, use the SD API routine `SDselect`. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `select` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

If `SDselect` finds the specified data set in the file, it returns an HDF4 SD data set identifier, called `sds_id` in the example. If it cannot find the data set, it returns `-1`.

Note Do not confuse HDF4 SD *file* identifiers, named `sd_id` in the examples, with HDF4 SD *data set* identifiers, named `sds_id` in the examples.

```
sds_id = hdfsd('select',sd_id,1)
```

Step 5: Getting Information About a Data Set. To read a data set, you must get information about the data set, such as its name, size, and data type. In the HDF4 SD API, you use the `SDgetinfo` routine to gather this information. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `getinfo` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

This code retrieves information about the data set identified by `sds_id`:

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] =
              hdfsd('getinfo',sds_id)
dsname =
      A
```

```
dsndims =
      2

dsdims =
      5     3

dstype =
      double

dsatts =
      0

stat =
      0
```

Step 6: Reading Data from the HDF4 File. To read data from an HDF4 file, you must use the `SDreaddata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API function, `readdata` in this case.
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`.
- Location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector. To read from the beginning of a data set, specify zero for each element of the start vector. Use `SDgetinfo` to determine the dimensions of the data set.
- Number of elements along each dimension to skip between each read operation, specified as a vector of scalar values, called the *stride* vector. To read every element of a data set, specify `1` as the value for each element of the vector or specify an empty array (`[]`).
- Total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector. To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set. Use `SDgetinfo` to determine these sizes.

Note `SDgetinfo` returns dimension values in row-major order, the ordering used by HDF4. Because MATLAB stores data in column-major order, you must specify the dimensions in column-major order, that is, `[columns,rows]`. In addition, you must use zero-based indexing in these arguments.

For example, to read the entire contents of a data set, use this code:

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] =
hdfsd('getinfo',sds_id);

ds_start = zeros(1,ds_ndims); % Creates the vector [0 0]
ds_stride = [];
ds_edges = ds_dims;

[ds_data, status] =
            hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
```

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read. For example, this code reads the entire second row of the sample data set:

```
ds_start = [0 1]; % Start reading at the first column, second row
ds_stride = []; % Read each element
ds_edges = [5 1]; % Read a 1-by-5 vector of data

[ds_data, status] =
           hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);
```

Step 7: Closing the HDF4 Data Set. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `endaccess` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

For example, this code closes the data set:

```
stat = hdfsd('endaccess',sds_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Step 8: Closing the HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `end` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

For example, this code closes the data set:

```
stat = hdfsd('end',sd_id);
```

Example: Exporting Data Using the HDF4 SD API Functions

The following sections provide a step-by-step example of how to export data from the MATLAB workspace to an HDF4 file using Scientific Data (SD) API functions.

- "Step 1: Creating an HDF4 File" on page 7-65
- "Step 2: Creating an HDF4 Data Set" on page 7-65
- "Step 3: Writing MATLAB Data to an HDF4 File" on page 7-67
- "Step 4: Writing Metadata to an HDF4 File" on page 7-69
- "Step 5: Closing HDF4 Data Sets" on page 7-70
- "Step 6: Closing an HDF4 File" on page 7-71

Step 1: Creating an HDF4 File. To export MATLAB data in HDF4 format, you must first create an HDF4 file, or open an existing one. In the HDF4 SD API, you use the `SDstart` routine. In MATLAB, use the `hdfsd` function, specifying `start` as the first argument. As other arguments, specify

- A text string specifying the name you want to assign to the HDF4 file (or the name of an existing HDF4 file)
- A text string specifying the HDF4 SD interface file access mode

For example, this code creates an HDF4 file named `mydata.hdf`:

```
sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');
```

When you specify the `DFACC_CREATE` access mode, `SDstart` creates the file and initializes the HDF4 SD multifile interface, returning an HDF4 SD file identifier, named `sd_id` in the example.

If you specify `DFACC_CREATE` mode and the file already exists, `SDstart` fails, returning `-1`. To open an existing HDF4 file, you must use HDF4 read or write modes. For information about using `SDstart` in these modes, see “Step 1: Opening the HDF4 File” on page 7-59.

Step 2: Creating an HDF4 Data Set. After creating the HDF4 file, or opening an existing one, you must create a data set in the file for each MATLAB array you want to export. If you are writing data to an existing data set, you can skip ahead to the next step.

In the HDF4 SD API, you use the `SDcreate` routine to create data sets. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'create'` in this case
- Valid HDF4 SD file identifier, `sd_id`, returned by `SDstart`
- Name you want assigned to the data set
- Data type of the data set.
- Number of dimensions in the data set. This is called the *rank* of the data set in HDF4 terminology.
- Size of each dimension, specified as a vector

Once you create a data set, you cannot change its name, data type, or dimensions.

For example, to create a data set in which you can write the following MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ];
```

you could call hdfsd, specifying as arguments 'create' and a valid HDF file identifier, sd_id. In addition, set the values of the other arguments as in this code fragment:

```
ds_name = 'A';
ds_type = 'double';
ds_rank = ndims(A);
ds_dims = fliplr(size(A));

sds_id = hdfsd('create',sd_id,ds_name,ds_type,ds_rank,ds_dims);
```

If SDcreate can successfully create the data set, it returns an HDF4 SD data set identifier, (sds_id). Otherwise, SDcreate returns -1.

In this example, note the following:

- The data type you specify in ds_type must match the data type of the MATLAB array that you want to write to the data set. In the example, the array is of class double so the value of ds_type is set to 'double'. If you wanted to use another data type, such as uint8, convert the MATLAB array to use this data type,

  ```
  A = uint8([ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ]);
  ```

  and specify the name of the MATLAB data type, uint8 in this case, in the ds_type argument.

  ```
  ds_type = 'uint8';
  ```

- The code fragment reverses the order of the values in the dimensions argument (ds_dims). This processing is necessary because the MATLAB size function returns the dimensions in column-major order and HDF4 expects to receive dimensions in row-major order.

Step 3: Writing MATLAB Data to an HDF4 File. After creating an HDF4 file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF4 SD API, you use the `SDwritedata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'writedata'` in this case
- Valid HDF4 SD data set identifier, `sds_id`, returned by `SDcreate`
- Location in the data set where you want to start writing data, called the *start* vector in HDF4 terminology
- Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF4 terminology
- Total number of elements to write along each dimension, called the *edges* vector in HDF4 terminology
- MATLAB array to be written

Note You must specify the values of the start, stride, and edges arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by the `size` function before assigning it as the value of the edges argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, the following code fragment writes this MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ];
```

into an HDF4 file:

```
ds_start = zeros(1:ndims(A)); % Start at the beginning
ds_stride = [];               % Write every element.
ds_edges = fliplr(size(A));   % Reverse the dimensions.

stat = hdfsd('writedata',sds_id,...
                          ds_start, ds_stride, ds_edges, A);
```

If it can write the data to the data set, `SDwritedata` returns `0`; otherwise, it returns `-1`.

Note `SDwritedata` queues write operations. To ensure that these queued write operations are executed, you must close the file, using the `SDend` routine. See "Step 6: Closing an HDF4 File" on page 7-71 for more information. As a convenience, MATLAB provides a function, `MLcloseall`, that you can use to close all open data sets and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-71 for more information.

To write less than the entire data set, use the start, stride, and edges vectors to specify where you want to start writing data and how much data you want to write.

For example, the following code fragment uses `SDwritedata` to replace the values of the entire second row of the sample data set:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

with the vector `B`:

```
B = [ 9 9 9 9 9];
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF4 uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as `(2,1)`. The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1]; % Start writing at the first column, second row.
ds_stride = []; % Write every element.
ds_edges = [5 1]; % Each row is a 1-by-5 vector.

stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Step 4: Writing Metadata to an HDF4 File. You can optionally include information in an HDF4 file, called attributes, that describes the file and its contents. Using the HDF4 SD API, you can associate attributes with three types of HDF4 objects:

- An entire HDF4 file — File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF4 file — Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set — Dimension attributes provide information about one particular dimension of a data set.

To create an attribute in the HDF4 SD API, use the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `'setattr'` as the first argument. As other arguments, specify

- A valid HDF4 SD identifier associated with the object. This value can be a file identifier (`sd_id`), a data set identifier (`sds_id`), or a dimension identifier (`dim_id`).
- A text string that defines the name of the attribute.
- The attribute value.

For example, this code creates a global attribute, named `my_global_attr`, and associates it with the HDF4 file identified by `sd_id`:

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the `SDsetattr` routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF4 function can determine the data type and size of the attribute from the value you specify.

The SD interface supports predefined attributes that have reserved names and, in some cases, data types. Predefined attributes are identical to user-defined attributes except that the HDF4 SD API has already defined

their names and data types. For example, the HDF4 SD API defines an attribute, named cordsys, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings 'cartesian', 'polar', and 'spherical'.

Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF4 SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA documentation.

In the HDF4 SD API, you create predefined attributes the same way you create user-defined attributes, using the SDsetattr routine. In MATLAB, use the hdfsd function, specifying setattr as the first argument:

```
attr_name = 'cordsys';
attr_value = 'polar';

status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF4 SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as SDsetdatastrs, are sometimes easier to use, especially when you are reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

You can associate multiple attributes with a single HDF4 object. HDF4 maintains an attribute index for each object. The attribute index is zero-based. The first attribute has index value 0, the second has index value 1, and so on. You access an attribute by its index value.

Each attribute has the format name=value, where name (called label in HDF4 terminology) is a text string up to 256 characters in length and value contains one or more entries of the same data type. A single attribute can have multiple values.

Step 5: Closing HDF4 Data Sets. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the SDendaccess routine to close a data set. In MATLAB, use the hdfsd function, specifying endaccess as the first argument. As the only other argument, specify a valid HDF4 SD data set identifier, sds_id in this example:

```
stat = hdfsd('endaccess',sds_id);
```

Step 6: Closing an HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying `end` as the first argument. As the only other argument, specify a valid HDF4 SD file identifier, `sd_id` in this example:

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Note Closing an HDF4 file executes all the write operations that have been queued using `SDwritedata`. As a convenience, the MATLAB HDF Utility API provides a function that can close all open data set and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-71 for more information.

Using the MATLAB HDF4 Utility API

In addition to the standard HDF4 APIs, listed in the `hdf`reference page, MATLAB supports utility functions that are designed to make it easier to use HDF4 in the MATLAB environment.

For example, using the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specifying the name of the `listinfo` routine as an argument, you can view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF identifiers are created or closed. In the following example only two identifiers are open.

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
 262144
```

```
Open scientific data file identifiers:
 393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Closing All Open HDF4 Identifiers. To close all the currently open HDF4 identifiers in a single call, use the gateway function to the MATLAB HDF4 utility API, `hdfml`, specifying the name of the `closeall` routine as an argument. The following example closes all the currently open HDF4 identifiers.

```
hdfml('closeall')
```

Error Handling

Error Reporting in MATLAB (p. 8-2)	The default error-reporting mechanism used by MATLAB
Capturing Information About the Error (p. 8-5)	Transferring information about an error using an object of the MException class
Throwing an Exception (p. 8-16)	Detecting a faulty condition in your application and throwing an exception
Responding to an Exception (p. 8-17)	Responding to an exception received by your program
Warnings (p. 8-22)	Identifying warnings and what caused them
Warning Control (p. 8-24)	Controlling the action taken when a warning is encountered
Debugging Errors and Warnings (p. 8-34)	Stopping code execution in the debugger on the occurrence of an error or warning

Error Reporting in MATLAB

In this section...

"Overview" on page 8-2

"Getting an Exception at the Command Line" on page 8-2

"Getting an Exception in Your Program Code" on page 8-3

"Generating a New Exception" on page 8-4

Overview

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when executed under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

In MATLAB, you can decide how your programs respond to different types of errors. You may want to prompt the user for more input, display extended error or warning information, or perhaps repeat a calculation using default values. The error-handling capabilities in MATLAB help your programs check for particular error conditions and execute the appropriate code depending on the situation.

When MATLAB detects a severe fault in the command or program it is running, it collects information about what was happening at the time of the error, displays a message to help the user understand what went wrong, and terminates the command or program. This is called *throwing an exception*. You can get an exception while entering commands at the MATLAB command prompt or while executing your program code.

Getting an Exception at the Command Line

If you get an exception at the MATLAB prompt, you have several options on how to deal with it as described below.

Determine the Fault from the Error Message

Evaluate the error message MATLAB has displayed. Most error messages attempt to explain at least the immediate cause of the program failure. There

is often sufficient information to determine the cause and what you need to do to remedy the situation.

Review the Failing Code

If the function in which the error occurred is implemented as an M-file, the error message should include a line that looks something like this:

```
surf

??? Error using ==> surf at 50
Not enough input arguments.
```

The underlined text to the right names the function that threw the error (`surf`, in this case) and shows the failing line number within that function's M-file. Click the underlined text; MATLAB opens the M-file and positions the cursor at the location in the file where the error originated. You may be able to determine the cause of the error by examining this line and the code that precedes it.

Step Through the Code in the Debugger

You can use the MATLAB Debugger to step through the failing code. Click the underlined error text to open the M-file in the MATLAB Editor at or near the point of the error. Next, click the hyphen at the beginning of that line to set a breakpoint at that location. When you rerun your program, MATLAB pauses execution at the breakpoint and enables you to step through the program code. The command `dbstop on error` is also helpful in finding the point of error.

See the documentation on "Editing and Debugging M-Files" for more information.

Getting an Exception in Your Program Code

When you are writing your own program in an M-file, you can *catch* exceptions and attempt to handle or resolve them instead of allowing your program to terminate. When you catch an exception, you interrupt the normal termination process and enter a block of code that deals with the faulty situation. This block of code is called a *catch block*.

Some of the things you might want to do in the catch block are:

- Examine information that has been captured about the error.
- Gather further information to report to the user.
- Try to accomplish the task at hand in some other way.
- Clean up any unwanted side effects of the error.

When you reach the end of the catch block, you can either continue executing the program, if possible, or terminate it.

The documentation on "Capturing Information About the Error" on page 8-5 describes how to acquire information about what caused the error, and "Responding to an Exception" on page 8-17 presents some ideas on how to respond to it.

Generating a New Exception

When your program code detects a condition that will either make the program fail or yield unacceptable results, it should throw an exception. This procedure

- Saves information about what went wrong and what code was executing at the time of the error.
- Gathers any other pertinent information about the error.
- Instructs MATLAB to throw the exception.

The documentation on "Capturing Information About the Error" on page 8-5 describes how to use an `MException` object to capture information about the error, and "Throwing an Exception" on page 8-16 explains how to initiate the exception process.

Capturing Information About the Error

In this section...

"Overview" on page 8-5

"The MException Class" on page 8-5

"Properties of the MException Class" on page 8-7

"Methods of the MException Class" on page 8-14

Overview

When MATLAB throws an exception, it captures information about what caused the error in a data structure called an `MException` object. This object is an instance of the MATLAB `MException` class. You can obtain access to the `MException` object by *catching* the exception before your program aborts and accessing the object constructed for this particular error via the `catch` command. When throwing an exception in response to an error in your own M-file code, you will have to create a new `MException` object and store information about the error in that object.

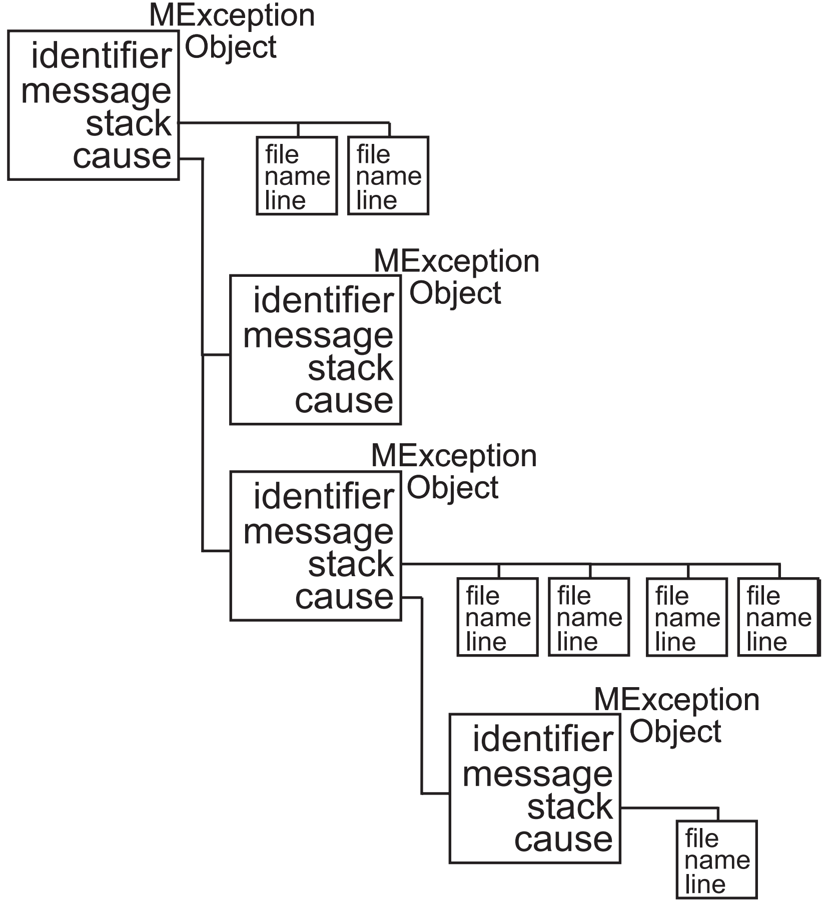
This section describes the `MException` class and objects constructed from that class:

Information on how to use this class is presented in later sections on "Responding to an Exception" on page 8-17 and "Throwing an Exception" on page 8-16.

The MException Class

The figure shown below illustrates one possible configuration of an object of the `MException` class. The object has four properties: `identifier`, `message`, `stack`, and `cause`. Each of these properties is implemented as a field of the structure that represents the `MException` object. The `stack` field is an N-by-1 array of additional structures, each one identifying an M-file, function, and line number from the call stack. The `cause` field is an M-by-1 cell array of `MException` objects, each representing an exception that is related to the current one.

See “Properties of the MException Class” on page 8-7 for a full description of these properties.



Object Constructor

Any code that detects an error and throws an exception must also construct an MException object in which to record and transfer information about the error. The syntax of the MException constructor is

```
ME = MException(identifier, message)
```

where `identifier` is a MATLAB message identifier of the form

```
component:mnemonic
```

that is enclosed in single quotes, and `message` is a text string, also enclosed in single quotes, that describes the error. The output `ME` is the resulting `MException` object.

If you are responding to an exception rather than throwing one, you do not have to construct an `MException` object. The object has already been constructed and populated by the code that originally detected the error.

Properties of the MException Class

The `MException` class has four properties. Each of these properties is implemented as a field of the structure that represents the `MException` object. Each of these properties is described in the sections below and referenced in the sections on "Responding to an Exception" on page 8-17 and "Throwing an Exception" on page 8-16:

- `identifier`
- `message`
- `stack`
- `cause`

Repeating the `surf` example shown above, but this time catching the exception, you can see the four properties of the `MException` object structure. (This example uses `try-catch` in an atypical fashion. See the section on "The try-catch Statement" on page 8-17 for more information on using `try-catch`).

```
try
    surf
catch ME
    ME
end
```

Run this at the command line and MATLAB returns the contents of the `MException` object:

```
ME =
 MException object with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         stack: [1x1 struct]
         cause: {}
```

The stack field shows the filename, function, and line number where the exception was thrown:

```
ME.stack
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The cause field is empty in this case. Each field is described in more detail in the sections that follow.

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs.

The message identifier is a read-only character string that specifies a *component* and a *mnemonic* label for an error or warning. The format of a simple identifier is

```
component:mnemonic
```

A colon separates the two parts of the identifier: component and mnemonic. If the identifier uses more than one mnemonic, then additional colons are required to separate them. A message identifier must always contain at least one colon.

Some examples of message identifiers are

```
MATLAB:divideByZero
Simulink:actionNotTaken
TechCorp:OpenFile:notFoundInPath
```

Both the `component` and `mnemonic` fields must adhere to the following syntax rules:

- No white space (space or tab characters) is allowed anywhere in the identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or an underscore.

There is no length limitation to either the `component` or `mnemonic`. The identifier can also be an empty string.

Component Field. The `component` field specifies a broad category under which various errors and warnings can be generated. Common components are a particular product or toolbox name, such as `MATLAB` or `Control`, or perhaps the name of your company, such as `TechCorp` in the preceding example.

You can also use this field to specify a multilevel component. The following statement has a three-level component followed by a mnemonic label:

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

The component field enables you to guarantee the uniqueness of each identifier. Thus, while MATLAB uses the identifier `MATLAB:divideByZero` for its `'Divide by zero'` warning, you can reuse the `divideByZero` mnemonic by using your own unique component. For example,

```
warning('TechCorp:divideByZero', ...
        'A sprocket value was divided by zero.')
```

Mnemonic Field. The `mnemonic` field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a simple component and mnemonic such as the following might be appropriate:

```
MATLAB:ambiguousSyntax
```

Message Identifiers in an MException Object. When throwing an exception, create an appropriate identifier and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

For example,

```
ME = MException('AcctError:Incomplete', ...
      'Client name not recognized.');

ME.identifier
ans =
    AcctError:NoClient
```

When responding to an exception, you can extract the message identifier from the `MException` object as shown here:

```
try
    surf
catch ME
    id = ME.identifier
end

id =
    MATLAB:nargchk:notEnoughInputs
```

Text of the Error Message

An error message in MATLAB is a read-only character string issued by the program code and returned in the `MException` object. This message can assist the user in determining the cause, and possibly the remedy, of the failure.

When throwing an exception, compose an appropriate error message and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

If your message string requires formatting specifications, like those available with the sprintf function, use this syntax for the MException constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts';  f1 = 'ClientName';
ME = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', S, f1);

ME.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

When responding to an exception, you can extract the error message from the MException object as follows:

```
try
    surf
catch ME
    msg = ME.message
end

msg =
    Not enough input arguments.
```

The Call Stack

The stack field of the MException object identifies the line number, function, and filename where the error was detected. If the error occurs in a called function, as in the following example, the stack field contains the line number, function name, and filename not only for the location of the immediate error, but also for each of the calling functions. In this case, stack is an N-by-1 array, where N represents the depth of the call stack. That is, the stack field displays the M-file function name and line number where the exception occurred, the name and line number of the M-file caller, the caller's caller, etc., until the top-most M-file function is reached.

When throwing an exception, MATLAB stores call stack information in the `stack` field. You cannot write to this field; access is read-only.

For example, suppose you have three functions that reside in two separate M-files:

```
 mfileA.m
=========================
        .
        .
42 function A1(x, y)
43 B1(x, y);



 mfileB.m
=========================
        .
        .
 8 function B1(x, y)
 9 B2(x, y)
        .
        .
26 function B2(x, y)
27      .
28      .
29      .
30      .
31 %  Throw exception here
```

Catch the exception in variable `ME` and then examine the `stack` field:

```
for k=1:length(ME.stack)
    ME.stack(k)
end

ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B2'
    line: 31
```

```
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B1'
    line: 9
ans =
    file: 'C:\matlab\test\mfileA.m'
    name: 'A1'
    line: 43
```

The Cause Array

In some situations, it can be important to record information about not only the one command that caused execution to stop, but also other exceptions that your code caught. You can save these additional `MException` objects in the `cause` field of the primary exception.

The `cause` field of an `MException` is an optional cell array of related `MException` objects. You must use the following syntax when adding objects to the `cause` cell array:

```
primaryException = addCause(primaryException, secondaryException)
```

This example attempts to assign an array `D` to variable `X`. If the `D` array does not exist, the code attempts to load it from a MAT-file and then retries assigning it to `X`. If the load fails, a new `MException` object (`ME3`) is constructed to store the cause of the first two errors (`ME1` and `ME2`):

```
try
    X = D(1:25)
catch ME1
    try
        filename = 'test200';
        load(filename);
        X = D(1:25)
    catch ME2
        ME3 = MException('MATLAB:LoadErr', ...
              'Unable to load from file %s', filename);
        ME3 = addCause(ME3, ME1);
        ME3 = addCause(ME3, ME2);
    end
end
```

There are two exceptions in the cause field of ME3:

```
ME3.cause
ans =
    [1x1 MException]
    [1x1 MException]
```

Examine the cause field of ME3 to see the related errors:

```
ME3.cause{:}
ans =

 MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
       message: 'Undefined function or method 'D' for input
arguments of type 'double'.'
         stack: [0x1 struct]
         cause: {}
ans =

 MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
       message: 'Unable to read file test204: No such file or
directory.'
         stack: [0x1 struct]
         cause: {}
```

Methods of the MException Class

There are ten methods that you can use with the MException class. The names of these methods are case-sensitive. See the MATLAB function reference pages for more information.

Method Name	Description
addCause	Append an MException to the cause field of another MException.
disp	Display an MException object.

Method Name	Description
eq	Compare MException objects for equality.
getReport	Return a formatted message based on the current exception.
isequal	Compare MException objects for equality.
last	Return the last uncaught exception. This is a static method.
ne	Compare MException objects for inequality.
rethrow	Reissue an exception that has previously been caught.
throw	Issue an exception.
throwAsCaller	Issue an exception, but omit the current stack frame from the stack field.

Throwing an Exception

When your program detects a fault that will keep it from completing as expected or will generate erroneous results, you should halt further execution and report the error by throwing an exception. The basic steps to take are

- Detect the error. This is often done with some type of conditional statement, such as an `if` statement that checks the output of the current operation.
- Construct an `MException` object to represent the error. Add a message identifier string and error message string to the object when calling the constructor.
- If there are other exceptions that may have contributed to the current error, you can store the `MException` object for each in the `cause` field of a single `MException` that you intend to throw. Use the `addCause` method for this.
- Use the `throw` or `throwAsCaller` function to have MATLAB issue the exception. At this point, MATLAB stores call stack information in the `stack` field of the `MException`, exits the currently running function, and returns control to either the keyboard or an enclosing catch block in a calling function.

Responding to an Exception

In this section...

"Overview" on page 8-17
"The try-catch Statement" on page 8-17
"Suggestions on How to Handle an Exception" on page 8-19

Overview

As stated earlier, MATLAB by default, terminates the currently running program when an exception is thrown. If you catch the exception in your program, however, you can capture information about what went wrong, and deal with the situation in a way that is appropriate for the particular condition. This requires a `try-catch` statement.

This section covers the following topics:

The try-catch Statement

When you have statements in your code that could generate undesirable results, put those statements into a `try-catch` block that catches any errors and handles them appropriately.

A `try-catch` statement looks something like the following pseudocode. It consists of two parts:

- A `try` block that includes all lines between the `try` and `catch` statements.
- A `catch` block that includes all lines of code between the `catch` and `end` statements.

```
     try
        Perform one ...
           or more operations
A    catch ME
        Examine error info in exception object ME
        Attempt to figure out what went wrong
        Either attempt to recover, or clean up and abort
```

```
    end

B   Program continues
```

The program executes the statements in the `try` block. If it encounters an error, it skips any remaining statements in the `try` block and jumps to the start of the `catch` block (shown here as point `A`). If all operations in the `try` block succeed, then execution skips the `catch` block entirely and goes to the first line following the `end` statement (point `B`).

Specifying the `try`, `catch`, and `end` commands and also the code of the `try` and `catch` blocks on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The Try Block

On execution, your code enters the `try` block and executes each statement as if it were part of the regular program. If no errors are encountered, MATLAB skips the `catch` block entirely and continues execution following the `end` statement. If any of the `try` statements fail, MATLAB immediately exits the `try` block, leaving any remaining statements in that block unexecuted, and enters the `catch` block.

The Catch Block

The `catch` command marks the start of a `catch` block and provides access to a data structure that contains information about what caused the exception. This is shown as the variable `ME` in the preceding pseudocode. This data structure is an object of the MATLAB `MException` class. When an exception occurs, MATLAB constructs an instance of this class and returns it in the `catch` statement that handles that error.

You are not required to specify any argument with the `catch` statement. If you do not need any of the information or methods provided by the `MException` object, just specify the `catch` keyword alone.

The MException object is constructed by internal code in the program that fails. The object has properties that contain information about the error that can be useful in determining what happened and how to proceed. The MException object also provides access to methods that enable you to respond to the exception. See the section on"The MException Class" on page 8-5 to find out more about the MException class.

Having entered the catch block, MATLAB executes the statements in sequence. These statements can attempt to

- Attempt to resolve the error.
- Capture more information about the error.
- Switch on information found in the MException object and respond appropriately.
- Clean up the environment that was left by the failing code.

The catch block often ends with a rethrow command. The rethrow causes MATLAB to exit the current function, keeping the call stack information as it was when the exception was first thrown. If this function is at the highest level, that is, it was not called by another function, the program terminates. If the failing function was called by another function, it returns to that function. Program execution continues to return to higher level functions, unless any of these calls were made within a higher-level try block, in which case the program executes the respective catch block.

More information about the MException class is provided in the section "Capturing Information About the Error" on page 8-5.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. The try block attempts to open and read the file. If either the open or read fails, the program catches the resulting exception and saves the MException object in the variable ME1.

The catch block in the example checks to see if the specified file could not be found. If so, the program allows for the possibility that a common variation of the filename extension (e.g., jpeg instead of jpg) was used by retrying

the operation with a modified extension. This is done using a try-catch statement nested within the original try-catch.

```
function d_in = read_image(filename)
file_format = regexp(filename, '(?<=\.)\w+$', 'match');

try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch file_format
      case 'jpg'    % Change jpg to jpeg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpeg');
      case 'jpeg'   % Change jpeg to jpg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpg');
      case 'tif'    % Change tif to tiff
         filename = regexprep(filename, '(?<=\.)\w+$', 'tiff');
      case 'tiff'   % Change tiff to tif
         filename = regexprep(filename, '(?<=\.)\w+$', 'tif');
      otherwise
         rethrow(ME1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch ME2
         ME2 = addCause(ME2, ME1);
         rethrow(ME2)
      end
   end
```

```
end
```

This example illustrates some of the actions that you can take in response to an exception:

- Compare the `identifier` field of the `MException` object against possible causes of the error.
- Use a nested `try-catch` statement to retry the open and read operations using a known variation of the filename extension.
- Display an appropriate message in the case that the file truly does not exist and then rethrow the exception.
- Add the first `MException` object to the `cause` field of the second.
- Rethrow the exception. This stops program execution and displays the error message.

Cleaning up any unwanted results of the error is also advisable. For example, your program may have allocated a significant amount of memory that it no longer needs.

Warnings

In this section...

"Reporting a Warning" on page 8-22
"Identifying the Cause" on page 8-23

Reporting a Warning

Like `error`, the `warning` function alerts the user of unexpected conditions detected when running a program. However, `warning` does not halt the execution of the program. It displays the specified warning message and then continues.

Use `warning` in your code to generate a warning message during execution. Specify the message string as the input argument to `warning`. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you do not want to see. You do this by invoking `warning` with certain control parameters. See "Warning Control" on page 8-24 for more information.

Formatted Message Strings

The warning message string you specify can contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the warning string the first argument, and add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters like `%d` and `%s` in the warning message string only when you specify more than one input argument with `warning`. See "Formatted Message Strings" on page 8-22 for information.

Message Identifiers

Use a message identifier argument with `warning` to attach a unique tag to a warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

See "Warning Control Statements" on page 8-26 for more information on how to use identifiers with warnings.

Identifying the Cause

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. Use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here:

```
lastwarn('newwarnmsg');   % Replace last warning with new string
lastwarn('');            % Replace last warning with empty string
```

Warning Control

In this section...

"Overview" on page 8-24

"Warning Statements" on page 8-25

"Warning Control Statements" on page 8-26

"Output from Control Statements" on page 8-28

"Saving and Restoring State" on page 8-30

"Backtrace and Verbose Modes" on page 8-31

Overview

MATLAB gives you the ability to control what happens when a warning is encountered during M-file program execution. Options that are available include

- Display selected warnings.
- Ignore selected warnings.
- Stop in the debugger when a warning is invoked.
- Display an M-stack trace after a warning is invoked.

Depending on how you set your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

1. Start by determining the scope of the control you need for the warnings generated by your code. Do you want the control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?

2. If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to each of those warnings. If, on the other

hand, you do not require that fine a granularity of control, the warning statements in your code need no message identifiers.

3 When you are ready to run your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements you put into your M-file code must contain the string to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you do not need to single out certain warnings for control, you need to specify only the message string. Use the syntax shown in "Warnings" on page 8-22. Valid formats are

```
warning('warnmsg')
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement

If you want to be able to apply control statements to specific warnings, you need to include a message identifier in the warning statements you wish to control. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See "Message Identifiers" on page 8-8 for information on how to specify the `msg_id` argument.

Note When you specify more than one input argument with `warning`, MATLAB treats the `warnmsg` string as if it were a `formatted_warnmsg`. This is explained in "Formatted Message Strings" on page 8-22.

Warning Control Statements

Once you have the warning statements in your M-file and are ready to execute it, you tell MATLAB how to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can return information on the state of selected warnings if you assign the output to a variable, as shown below. See “Output from Control Statements” on page 8-28.

```
s = warning('state', 'msg_id');
```

Warning States

There are three possible values for the `state` argument of a warning control statement.

State	Description
`on`	Enable the display of selected warning message.
`off`	Disable the display of selected warning message.
`query`	Display the current state of selected warning.

Message Identifiers

In addition to the message identifiers already discussed, there are three other identifiers that you can use in control statements only.

Identifier	Description
msg_id string	Set selected warning to the specified state.
`all`	Set all warnings to the specified state.
`last`	Set only the last displayed warning to the specified state.

Note MATLAB starts up with all warnings enabled, except for those displayed in response to the command, `warning('query', 'all')`.

Example 1 — Enabling a Selected Warning

Enable just the `actionNotTaken` warning from Simulink by first turning off all warnings and then setting just that warning to `on`.

```
warning off all
warning on Simulink:actionNotTaken
```

Next, use `query` to determine the current state of all warnings. It reports that you have set all warnings to `off`, with the exception of `Simulink:actionNotTaken`.

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are

  State  Warning Identifier

     on  Simulink:actionNotTaken
```

Example 2 — Disabling the Most Recent Warning

Evaluating `inv` on zero displays a warning message. Turn off the most recently invoked warning with `warning off last`.

```
inv(0)
Warning: Matrix is singular to working precision.
ans =
   Inf

warning off last

inv(0)              % No warning is displayed this time
ans =
   Inf
```

Output from Control Statements

The `warning` function, when used in a control statement, returns a MATLAB structure array containing the previous state of the selected warning(s). Use the following syntax to return this information in structure array `s`:

```
s = warning('state', 'msg_id');
```

You must type the command using the MATLAB function format; parentheses and quotation marks are required.

Note MATLAB does not display `warning` output if you do not assign the output to a variable.

The next example turns off `divideByZero` warnings for the MATLAB component, and returns the `identifier` and previous `state` in a 1-by-1 structure array.

```
s = warning('off','MATLAB:divideByZero')
s =
    identifier: 'MATLAB:divideByZero'
         state: 'on'
```

You can use output variables with any type of warning control statement. If you just want to collect the information but do not want to change state, simply perform a `query` on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but save the former state so you can restore it later, use the return structure array to save that state. The following example does an implicit `query`, returning state information in `s`, and then turns on all warnings.

```
s = warning('on', 'all');
```

See “Saving and Restoring State” on page 8-30, for more information on restoring the former state of warnings.

Output Structure Array

Each element of the structure array returned by `warning` contains two fields.

Field Name	Description
`identifier`	Message identifier string, `'all'`, or `'last'`
`state`	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or `'last'` in the command, MATLAB returns a one-element structure array. The `identifier` field contains the selected message identifier, and the `state` field holds the current state of that warning:

```
s = warning('query','last')
s =
   identifier: 'MATLAB:divideByZero'
        state: 'on'
```

If you query for the state of all warnings, using `'all'` in the command, MATLAB returns a structure array having one or more elements:

- The first element of the array always represents the default state. (This is the state set by the last `warning on|off all` command.)
- Each other element of the array represents a warning that is in a state different from the default.

```
warning off all
warning on MATLAB:divideByZero
warning on MATLAB:fileNotFound

s = warning('query', 'all')
s =
   3x1 struct array with fields:
       identifier
       state

s(1)
ans =
```

```
      identifier: 'all'
           state: 'off'

   s(2)
   ans =
      identifier: 'MATLAB:divideByZero'
           state: 'on'

   s(3)
   ans =
      identifier: 'MATLAB:fileNotFound'
           state: 'on'
```

Saving and Restoring State

To temporarily change the state of some warnings and then later return to your original settings, save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in "Output from Control Statements" on page 8-28. The following statement saves the current state of all warnings in structure array s:

```
s = warning('query', 'all');
```

To restore state from s, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1 — Performing an Explicit Query

Perform a query of all warnings to save the current state in structure array s:

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

Example 2 — Performing an Implicit Query

Turn on one particular warning, saving the previous state of this warning in `s`. Remember that this nonquery syntax (where `state` equals `on` or `off`) performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Backtrace and Verbose Modes

In addition to warning messages, there are two *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
`backtrace`	Display an M-stack trace after a warning is invoked.	`on (enabled)`
`verbose`	Display a message on how to suppress the warning.	`off (terse)`

The syntax for this type of control statement is as follows, where `state`, in this case, can be only `on`, `off`, or `query`:

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Note You cannot save and restore the current state of the `backtrace` or `verbose` modes as you can with other states.

Example 1 — Displaying a Stack Trace on a Specific Warning

It can be difficult to locate the source of a warning when it is generated from code buried in several levels of function calls. This example generates

a warning within a function that is nested several levels deep within the primary function in file `f1.m`:

```
function f1(a, b)
   for k = a:-1:b
       f2(k)
   end
   function f2(x)
       f3(x-1)
       function f3(y)
           x = log(y);
       end
   end
end
```

After enabling all warnings, run the M-file. The code generates a `Log of zero` warning. In an M-file of this size, it is not difficult to find the cause of the warning, but in an M-file of several hundred lines, this could take some time:

```
warning on all

f1(50,1)
Warning: Log of zero.
```

To simplify the debug process, enable backtrace mode. In this mode, MATLAB reports which function generated the warning (`f3`), the line number of the attempted operation (line `8`), the sequence of function calls that led up to the execution of the function (`f1>f2/f3`), and the line at which each of these function call was made (`3` and `6`):

```
warning on backtrace
f1(50,1)
Warning: Log of zero.
> In f1>f2/f3 at 8
  In f1>f2 at 6
  In f1 at 3
```

Example 2 — Enabling Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it:

Turn on all warnings, disable backtrace (if you have just run the previous example), and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Call the function described in Example 1 to find out how to suppress any warnings generated by that function:

```
f1(50,1)
Warning: Log of zero.
(Type "warning off MATLAB:log:logOfZero" to suppress this warning.)
```

Use the message identifier `MATLAB:log:logOfZero` to disable only this warning, and run the function again. This time the warning message is not displayed:

```
warning off MATLAB:log:logOfZero

f1(50,1)
```

Debugging Errors and Warnings

You can direct MATLAB to temporarily stop the execution of an M-file in the event of a run-time error or warning, at the same time opening a debug window paused at the M-file line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the `dbstop` function to have MATLAB stop execution and enter debug mode when any M-file you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
`dbstop if all error`	Stop on any error.
`dbstop if error`	Stop on any error not detected within a `try-catch` block.
`dbstop if warning`	Stop on any warning.

In all three cases, the M-file you are trying to debug must be in a directory that is on the search path or in the current directory.

You cannot resume execution after an error; use `dbquit` to exit from the Debugger. To resume execution after a warning, use `dbcont` or `dbstep`.

9

Classes and Objects

Classes and Objects: An Overview (p. 9-2)	Using object-oriented programming in MATLAB
Designing User Classes in MATLAB (p. 9-9)	The basic set of methods that should be included in a class
Overloading Operators and Functions (p. 9-23)	Overloading the MATLAB operators and functions to change their behavior
Example — A Polynomial Class (p. 9-26)	Example that defines a new class to implement a MATLAB data type for polynomials
Building on Other Classes (p. 9-38)	Inheritance and aggregation
Example — Assets and Asset Subclasses (p. 9-41)	An example that uses simple inheritance
Example — The Portfolio Container (p. 9-58)	An example that uses aggregation
Saving and Loading Objects (p. 9-64)	Saving and retrieving user-defined objects to and from MAT-files
Example — Defining saveobj and loadobj for Portfolio (p. 9-65)	Defining methods that automatically execute on save and load
Object Precedence (p. 9-70)	Determining which operator or function to call in a given situation
How MATLAB Determines Which Method to Call (p. 9-72)	How function arguments and precedence determine which method to call

Classes and Objects: An Overview

In this section...

"Overview" on page 9-2
"Features of Object-Oriented Programming" on page 9-3
"MATLAB Data Class Hierarchy" on page 9-3
"Creating Objects" on page 9-4
"Invoking Methods on Objects" on page 9-4
"Private Methods" on page 9-5
"Helper Functions" on page 9-6
"Debugging Class Methods" on page 9-6
"Setting Up Class Directories" on page 9-6
"Data Structure" on page 9-7
"Tips for C++ and Java Programmers" on page 9-8

Overview

You can view classes as new data types having specific behaviors defined for the class. For example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations,

subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as *overloading* the operator.

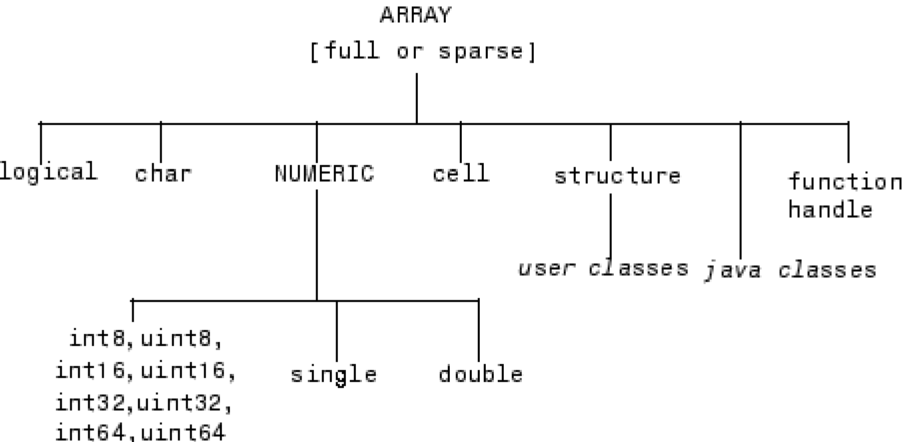
Features of Object-Oriented Programming

When using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend. Programming with classes and objects differs from ordinary structured programming in these important ways:

- **Function and operator overloading**. You can create methods that override existing MATLAB functions. When you call a function with a user-defined object as an argument, MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal MATLAB function.
- **Encapsulation of data and methods**. Object properties are not visible from the command line; you can access them only with class methods. This protects the object properties from operations that are not intended for the object's class.
- **Inheritance**. You can create class hierarchies of parent and child classes in which the child class inherits data fields and methods from the parent. A child class can inherit from one parent (*single inheritance*) or many parents (*multiple inheritance*). Inheritance can span one or more generations. Inheritance enables sharing common parent functions and enforcing common behavior amongst all child classes.
- **Aggregation**. You can create classes using *aggregation*, in which an object contains other objects. This is appropriate when an object type is part of another object type. For example, a savings account object might be a part of a financial portfolio object.

MATLAB Data Class Hierarchy

All MATLAB data types are designed to function as classes in object-oriented programming. The diagram below shows the fifteen fundamental data types (or classes) defined in MATLAB. You can add new data types to MATLAB by extending the class hierarchy.



The diagram shows a *user class* that inherits from the structure class. All classes that you create are structure based since this is the point in the class hierarchy where you can insert your own classes. (For more information about MATLAB data types, see Chapter 2, "Data Types")

Creating Objects

You create an object by calling the class constructor and passing it the appropriate input arguments. In MATLAB, constructors have the same name as the class name. For example, the statement,

```
p = polynom([1 0 -2 -5]);
```

creates an object named `p` belonging to the class `polynom`. Once you have created a polynom object, you can operate on the object using methods that are defined for the `polynom` class. See "Example — A Polynomial Class" on page 9-26 for a description of the `polynom` class.

Invoking Methods on Objects

Class methods are M-file functions that take an object as one of the input arguments. The methods for a specific class must be placed in the class directory for that class (the `@classname` directory). This is the first place that MATLAB looks to find a class method.

The syntax for invoking a method on an object is similar to a function call. Generally, it looks like

```
[out1,out2,...] = methodName(object,arg1,arg2, ...);
```

For example, suppose a user-defined class called `polynom` has a `char` method defined for the class. This method converts a polynom object to a character string and returns the string. This statement calls the `char` method on the polynom object `p`.

```
s = char(p);
```

Using the `class` function, you can confirm that the returned value `s` is a character string.

```
class(s)
ans =
    char

s
s =
    x^3-2*x-5
```

You can use the `methods` command to produce a list of all of the methods that are defined for a class.

Private Methods

Private methods can be called only by other methods of their class. You define private methods by placing the associated M-files in a `private` subdirectory of the `@classname` directory. In the example,

```
@classname/private/updateObj.m
```

the method `updateObj` has scope only within the *`classname`* class. This means that `updateObj` can be called by any method that is defined in the `@classname` directory, but it cannot be called from the MATLAB command line or by methods outside of the class directory, including parent methods.

Private methods and private functions differ in that private methods (in fact all methods) have an object as one of their input arguments and private

functions do not. You can use private functions as helper functions, such as described in the next section.

Helper Functions

In designing a class, you may discover the need for functions that perform support tasks for the class, but do not directly operate on an object. These functions are called *helper functions*. A helper function can be a subfunction in a class method file or a private function. When determining which version of a particular function to call, MATLAB looks for these functions in the order listed above. For more information about the order in which MATLAB calls functions and methods, see “How MATLAB Determines Which Method to Call” on page 9-72.

Debugging Class Methods

You can use the MATLAB debugging commands with object methods in the same way that you use them with other M-files. The only difference is that you need to include the class directory name before the method name in the command call, as shown in this example using `dbstop`.

```
dbstop @polynom/char
```

While debugging a class method, you have access to all methods defined for the class, including inherited methods, private methods, and private functions.

Changing Class Definition

If you change the class definition, such as the number or names of fields in a class, you must issue a

```
clear classes
```

command to propagate the changes to your MATLAB session. This command also clears all objects from the workspace. See the `clear` command help entry for more information.

Setting Up Class Directories

The M-files defining the methods for a class are collected together in a directory referred to as the class directory. The directory name is formed with

the class name preceded by the character @. For example, one of the examples used in this chapter is a class involving polynomials in a single variable. The name of the class, and the name of the class constructor, is `polynom`. The M-files defining a polynomial class would be located in directory with the name `@polynom`.

The class directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For instance, the new `@polynom` directory could be a subdirectory of the MATLAB working directory or your own personal directory that has been added to the search path.

Adding the Class Directory to the MATLAB Path

After creating the class directory, you need to update the MATLAB path so that MATLAB can locate the class source files. The class directory should not be directly on the MATLAB path. Instead, you should add the parent directory to the MATLAB path. For example, if the `@polynom` class directory is located at

```
c:\myClasses\@polynom
```

you add the class directory to the MATLAB path with the `addpath` command

```
addpath c:\myClasses;
```

Using Multiple Class Directories

A MATLAB class can access methods in multiple @*`classname`* directories if all such directories are visible to MATLAB (i.e., the parent directories are on the MATLAB path or in the current directory). When you attempt to use a method of the class, MATLAB searches all the visible directories named @*`classname`* for the appropriate method.

For more information, see "How MATLAB Determines Which Method to Call" on page 9-72.

Data Structure

One of the first steps in the design of a new class is the choice of the data structure to be used by the class. Objects are stored in MATLAB structures. The fields of the structure, and the details of operations on the fields, are

visible only within the methods for the class. The design of the appropriate data structure can affect the performance of the code.

Tips for C++ and Java Programmers

If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

- In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.
- In MATLAB, there is no equivalent to a destructor method. To remove an object from the workspace, use the `clear` function.
- Construction of MATLAB data types occurs at runtime rather than compile time. You register an object as belonging to a class by calling the `class` function.
- When using inheritance in MATLAB, the inheritance relationship is established in the child class by creating the parent object, and then calling the `class` function. For more information on writing constructors for inheritance relationships, see "Building on Other Classes" on page 9-38.
- When using inheritance in MATLAB, the child object contains a parent object in a property with the name of the parent class.
- In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For instance, this call to the `set` method updates the `name` field of the object `A` and returns the updated object.

  ```
  A = set(A,'name','John Smith');
  ```

- In MATLAB, there is no equivalent to an abstract class.
- In MATLAB, there is no equivalent to the C++ scoping operator.
- In MATLAB, there is no virtual inheritance or virtual base classes.
- In MATLAB, there is no equivalent to C++ templates.

Designing User Classes in MATLAB

In this section...

"The MATLAB Canonical Class" on page 9-9

"The Class Constructor Method" on page 9-10

"Examples of Constructor Methods" on page 9-12

"Identifying Objects Outside the Class Directory" on page 9-12

"The display Method" on page 9-13

"Accessing Object Data" on page 9-13

"The set and get Methods" on page 9-14

"Indexed Reference Using subsref and subsasgn" on page 9-15

"Handling Subscripted Reference" on page 9-16

"Handling Subscripted Assignment" on page 9-19

"Object Indexing Within Methods" on page 9-20

"Defining end Indexing for an Object" on page 9-20

"Indexing an Object with Another Object" on page 9-21

"Converter Methods" on page 9-22

The MATLAB Canonical Class

When you design a MATLAB class, you should include a standard set of methods that enable the class to behave in a consistent and logical way within the MATLAB environment. Depending on the nature of the class you are defining, you may not need to include all of these methods and you may include a number of other methods to realize the class's design goals.

This table lists the basic methods included in MATLAB classes.

Class Method	Description
class constructor	Creates an object of the class.

Class Method	Description
`display`	Called whenever MATLAB displays the contents of an object (e.g., when an expression is entered without terminating with a semicolon).
`set` and `get`	Accesses class properties.
`subsref and subsasgn`	Enables indexed reference and assignment for user objects.
`end`	Supports `end` syntax in indexing expressions using an object; e.g., `A(1:end)`.
`subsindex`	Supports using an object in indexing expressions.
converters like `double` and `char`	Methods that convert an object to a MATLAB data type.

The following sections discuss the implementation of each type of method, as well as providing references to examples used in this chapter.

The Class Constructor Method

The @ directory for a particular class must contain an M-file known as the *constructor* for that class. The name of the constructor is the same as the name of the directory (excluding the @ prefix and `.m` extension) that defines the name of the class. The constructor creates the object by initializing the data structure and instantiating an object of the class.

Guidelines for Writing a Constructor

Class constructors must perform certain functions so that objects behave correctly in the MATLAB environment. In general, a class constructor must handle three possible combinations of input arguments:

- No input arguments
- An object of the same class as an input argument
- The input arguments used to create an object of the class (typically data of some kind)

No Input Arguments. If there are no input arguments, the constructor should create a default object. Since there are no inputs, you have no data from which to create the object, so you simply initialize the object's data structures with empty or default values, call the `class` function to instantiate the object, and return the object as the output argument. Support for this syntax is required for two reasons:

- When loading objects into the workspace, the `load` function calls the class constructor with no arguments.
- When creating arrays of objects, MATLAB calls the class constructor to add objects to the array.

Object Input Argument. If the first input argument in the argument list is an object of the same class, the constructor should simply return the object. Use the `isa` function to determine if an argument is a member of a class. See "Overloading the + Operator" on page 9-32 for an example of a method that uses this constructor syntax.

Data Input Arguments. If the input arguments exist and are not objects of the same class, then the constructor creates the object using the input data. Of course, as in any function, you should perform proper argument checking in your constructor function. A typical approach is to use a `varargin` input argument and a `switch` statement to control program flow. This provides an easy way to accommodate the three cases: no inputs, object input, or the data inputs used to create an object.

It is in this part of the constructor that you assign values to the object's data structure, call the `class` function to instantiate the object, and return the object as the output argument. If necessary, place the object in an object hierarchy using the `superiorto` and `inferiorto` functions.

Using the class Function in Constructors

Within a constructor method, you use the `class` function to associate an object structure with a particular class. This is done using an internal class tag that is only accessible using the `class` and `isa` functions. For example, this call to the `class` function identifies the object `p` to be of type `polynom`.

```
p = class(p,'polynom');
```

Examples of Constructor Methods

See the following sections for examples of constructor methods:

- "The Polynom Constructor Method" on page 9-27
- "The Asset Constructor Method" on page 9-43
- "The Stock Constructor Method" on page 9-50
- "The Portfolio Constructor Method" on page 9-59

Identifying Objects Outside the Class Directory

The `class` and `isa` functions used in constructor methods can also be used outside of the class directory. The expression

```
isa(a,'classname');
```

checks whether `a` is an object of the specified class. For example, if `p` is a polynom object, each of the following expressions is true.

```
isa(pi,'double');
isa('hello','char');
isa(p,'polynom');
```

Outside of the class directory, the `class` function takes only one argument (it is only within the constructor that `class` can have more than one argument).

The expression

```
class(a)
```

returns a string containing the class name of `a`. For example,

```
class(pi),
class('hello'),
class(p)
```

return

```
'double',
'char',
'polynom'
```

Use the whos function to see what objects are in the MATLAB workspace.

```
whos

  Name      Size          Bytes  Class
  p         1x1           156 polynom object
```

The display Method

MATLAB calls a method named display whenever an object is the result of a statement that is not terminated by a semicolon. For example, creating the variable a, which is a double, calls the MATLAB display method for doubles.

```
a = 5
a =
    5
```

You should define a display method so MATLAB can display values on the command line when referencing objects from your class. In many classes, display can simply print the variable name, and then use the char converter method to print the contents or value of the variable, since MATLAB displays output as strings. You must define the char method to convert the object's data to a character string.

Examples of display Methods

See the following sections for examples of display methods:

- "The Polynom display Method" on page 9-30
- "The Asset display Method" on page 9-48
- "The Stock display Method" on page 9-57
- "The Portfolio display Method" on page 9-61

Accessing Object Data

You need to write methods for your class that provide access to an object's data. Accessor methods can use a variety of approaches, but all methods that change object data always accept an object as an input argument and return a new object with the data changed. This is necessary because MATLAB does

not support passing arguments by reference (i.e., pointers). Functions can change only their private, temporary copy of an object. Therefore, to change an existing object, you must create a new one, and then replace the old one.

The following sections provide more detail about implementation techniques for the `set`, `get`, `subsasgn`, and `subsref` methods.

The set and get Methods

The `set` and `get` methods provide a convenient way to access object data in certain cases. For example, suppose you have created a class that defines an arrow object that MATLAB can display on graphs (perhaps composed of existing MATLAB line and patch objects).

To produce a consistent interface, you could define `set` and `get` methods that operate on arrow objects the way the MATLAB `set` and `get` functions operate on built-in graphics objects. The `set` and `get` verbs convey what operations they perform, but insulate the user from the internals of the object.

Examples of set and get Methods

See the following sections for examples of `set` and `get` methods:

- "The Asset get Method" on page 9-44 and "The Asset set Method" on page 9-45
- "The Stock get Method" on page 9-52 and "The Stock set Method" on page 9-53

Property Name Methods

As an alternative to a general `set` method, you can write a method to handle the assignment of an individual property. The method should have the same name as the property name.

For example, if you defined a class that creates objects representing employee data, you might have a field in an employee object called `salary`. You could then define a method called `salary.m` that takes an employee object and a value as input arguments and returns the object with the specified value set.

Indexed Reference Using subsref and subsasgn

User classes implement new data types in MATLAB. It is useful to be able to access object data via an indexed reference, as is possible with the MATLAB built-in data types. For example, if `A` is an array of class `double`, `A(i)` returns the `i`th element of `A`.

As the class designer, you can decide what an index reference to an object means. For example, suppose you define a class that creates polynomial objects and these objects contain the coefficients of the polynomial.

An indexed reference to a polynomial object,

```
p(3)
```

could return the value of the coefficient of x^3, the value of the polynomial at $x = 3$, or something different depending on the intended design.

You define the behavior of indexing for a particular class by creating two class methods - `subsref` and `subsasgn`. MATLAB calls these methods whenever a subscripted reference or assignment is made on an object from the class. If you do not define these methods for a class, indexing is undefined for objects of this class.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. For details, see “Structures” on page 2-74.

Behavior Within Class Methods

If `A` is an array of one of the fundamental MATLAB data types, then referencing a value of `A` using an indexed reference calls the built-in MATLAB `subsref` method. It does not call any `subsref` method that you might have overloaded for that data type.

For example, if `A` is an array of type double, and there is an `@double/subsref` method on your MATLAB path, the statement `B = A(I)` does not call this method, but calls the MATLAB built-in `subsref` method instead.

The same is true for user-defined classes. Whenever a class method requires the functionality of the overloaded `subsref` or `subsassign`, it must call the

overloaded methods with function calls rather than using operators like '()', '{}', or '.'.

For example, suppose you define a polynomial class that defines a subsref method that causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. Therefore,

```
p = polynom([1 0 -2 -5]);
```

The following subscripted expression returns the value of the polynomial at

```
x = 3 and x = 4.
p([3 4])
ans =
    16   51
```

Now suppose that you want to use this feature in one of the class methods. To do so, you must call the subsref function directly:

```
y = polyval(p,x);
 subs.type = '()';
 subs.subs = {x};
 y = subsref(p, subs);  % Need to call subsref here
```

Handling Subscripted Reference

The use of a subscript or field designator with an object on the right-hand side of an assignment statement is known as a *subscripted reference*. MATLAB calls a method named subsref in these situations.

Object subscripted references can be of three forms — an array index, a cell array index, and a structure field name:

```
A(I)
A{I}
A.field
```

Each of these results in a call by MATLAB to the subsref method in the class directory. MATLAB passes two arguments to subsref.

```
B = subsref(A,S)
```

The first argument is the object being referenced. The second argument, `S`, is a structure array with two fields:

- `S.type` is a string containing `'()'`, `'{}'`, or `'.'` specifying the subscript type. The parentheses represent a numeric array; the curly braces, a cell array; and the dot, a structure array.
- `S.subs` is a cell array or string containing the actual subscripts. A colon used as a subscript is passed as a cell array containing the string `':'`.

For instance, the expression

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

causes MATLAB to call `subsref(A,S)`, where `S` is a 1-by-1 structure with

```
S.type = '()'
S.subs = {1:2,':'}
```

Similarly, the expression

```
A{1:2}
```

uses

```
S.type ='{}'
S.subs = {1:2}
```

The expression

```
A.field
```

calls `subsref(A,S)` where

```
S.type = '.'
S.subs = 'field'
```

These simple calls are combined for more complicated subscripting expressions. In such cases, `length(S)` is the number of subscripting levels. For example,

```
A(1,2).name(3:4)
```

calls `subsref(A,S)`, where `S` is a 3-by-1 structure array with the values:

```
S(1).type = '()'    S(2).type = '.'      S(3).type = '()'
S(1).subs = {1,2}   S(2).subs = 'name'   S(3).subs = {3:4}
```

How to Write subsref

The `subsref` method must interpret the subscripting expressions passed in by MATLAB. A typical approach is to use the `switch` statement to determine the type of indexing used and to obtain the actual indices. The following three code fragments illustrate how to interpret the input arguments. In each case, the function must return the value `B`.

For an array index:

```
switch S.type
case '()'
   B = A(S.subs{:});
end
```

For a cell array:

```
switch S.type
case '{}'
   B = A(S.subs{:});          % A is a cell array
end
```

For a structure array:

```
switch S.type
case '.'
   switch S.subs
   case 'field1'
      B = A.field1;
   case 'field2'
      B = A.field2;
   end
end
```

Examples of the subsref Method

See the following sections for examples of the `subsref` method:

- "The Polynom subsref Method" on page 9-31
- "The Asset subsref Method" on page 9-46
- "The Stock subsref Method" on page 9-54
- "The Portfolio subsref Method" on page 9-68

Handling Subscripted Assignment

The use of a subscript or field designator with an object on the left-hand side of an assignment statement is known as a *subscripted assignment*. MATLAB calls a method named `subsasgn` in these situations. Object subscripted assignment can be of three forms - an array index, a cell array index, and a structure field name.

```
A(I) = B
A{I} = B
A.field = B
```

Each of these results in a call to `subsasgn` of the form

```
A = subsasgn(A,S,B)
```

The first argument, `A`, is the object being referenced. The second argument, `S`, has the same fields as those used with `subsref`. The third argument, `B`, is the new value.

Examples of the subsasgn Method

See the following sections for examples of the `subsasgn` method:

- "The Asset subsasgn Method" on page 9-47
- "The Stock subsasgn Method" on page 9-55

Object Indexing Within Methods

If a subscripted reference is made within a class method, MATLAB uses its built-in subsref function to access data within the method's own class. If the method accesses data from another class, MATLAB calls the overloaded subsref function in that class. The same holds true for subscripted assignment and subsasgn.

The following example shows a method, testref, that is defined in the class, employee. This method makes a reference to a field, address, in an object of its own class. For this, MATLAB uses the built-in subsref function. It also references the same field in another class, this time using the overloaded subsref of that class.

```
% ---- EMPLOYEE class method: testref.m ----
function testref(myclass,otherclass)

myclass.address                    % use built-in subsref
otherclass.address                 % use overloaded subsref
```

The example creates an employee object and a company object.

```
empl = employee('Johnson','Chicago');
comp = company('The MathWorks','Natick');
```

The employee class method, testref, is called. MATLAB uses an overloaded subsref only to access data outside of the method's own class.

```
testref(empl,comp)
ans =                               % built-in subsref was called
   Chicago

ans =                               % @company\subsref was called
Executing @company\subsref ...
   Natick
```

Defining end Indexing for an Object

When you use end in an object indexing expression, MATLAB calls the object's end class method. If you want to be able to use end in indexing expressions involving objects of your class, you must define an end method for your class.

The end method has the calling sequence

```
end(a,k,n)
```

where a is the user object, k is the index in the expression where the end syntax is used, and n is the total number of indices in the expression.

For example, consider the expression

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

MATLAB calls the end method defined for the object A using the arguments

```
end(A,1,2)
```

That is, the end statement occurs in the first index element and there are two index elements. The class method for end must then return the index value for the last element of the first dimension. When you implement the end method for your class, you must ensure it returns a value appropriate for the object.

Indexing an Object with Another Object

When MATLAB encounters an object as an index, it calls the subsindex method defined for the object. For example, suppose you have an object a and you want to use this object to index into another object b.

```
c = b(a);
```

A subsindex method might do something as simple as convert the object to double format to be used as an index, as shown in this sample code.

```
function d = subsindex(a)
%SUBSINDEX
% convert the object a to double format to be used
% as an index in an indexing expression
d = double(a);
```

subsindex values are 0-based, not 1-based.

Converter Methods

A converter method is a class method that has the same name as another class, such as `char` or `double`. Converter methods accept an object of one class as input and return an object of another class. Converters enable you to

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly

A converter function call is of the form

```
b = classname(a)
```

where `a` is an object of a class other than *`classname`*. In this case, MATLAB looks for a method called *`classname`* in the class directory for object `a`. If the input object is already of type *`classname`*, then MATLAB calls the constructor, which just returns the input argument.

Examples of Converter Methods

See the following sections for examples of converter methods:

- "The Polynom to Double Converter" on page 9-28
- "The Polynom to Char Converter" on page 9-29

Overloading Operators and Functions

In this section...
"Overloading Operators" on page 9-23
"Overloading Functions" on page 9-25

Overloading Operators

In many cases, you may want to change the behavior of the MATLAB operators and functions for cases when the arguments are objects. You can accomplish this by overloading the relevant functions. Overloading enables a function to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest-precedence object. See "Object Precedence" on page 9-70 for more information on object precedence.

Each built-in MATLAB operator has an associated function name (e.g., the + operator has an associated `plus.m` function). You can overload any operator by creating an M-file with the appropriate name in the class directory. For example, if either `p` or `q` is an object of type `classname`, the expression

```
p + q
```

generates a call to a function `@classname/plus.m`, if it exists. If `p` and `q` are both objects of different classes, then MATLAB applies the rules of precedence to determine which method to use.

Examples of Overloaded Operators

See the following sections for examples of overloaded operators:

- "Overloading the + Operator" on page 9-32
- "Overloading the - Operator" on page 9-33
- "Overloading the * Operator" on page 9-33

The following table lists the function names for most of the MATLAB operators.

Operation	M-File	Description
a + b	plus(a,b)	Binary addition
a - b	minus(a,b)	Binary subtraction
-a	uminus(a)	Unary minus
+a	uplus(a)	Unary plus
a.*b	times(a,b)	Element-wise multiplication
a*b	mtimes(a,b)	Matrix multiplication
a./b	rdivide(a,b)	Right elementwise division
a.\b	ldivide(a,b)	Left elementwise division
a/b	mrdivide(a,b)	Matrix right division
a\b	mldivide(a,b)	Matrix left division
a.^b	power(a,b)	Element-wise power
a^b	mpower(a,b)	Matrix power
a < b	lt(a,b)	Less than
a > b	gt(a,b)	Greater than
a <= b	le(a,b)	Less than or equal to
a >= b	ge(a,b)	Greater than or equal to
a ~= b	ne(a,b)	Not equal to
a == b	eq(a,b)	Equality
a & b	and(a,b)	Logical AND
a | b	or(a,b)	Logical OR
~a	not(a)	Logical NOT
a:d:b a:b	colon(a,d,b) colon(a,b)	Colon operator
a'	ctranspose(a)	Complex conjugate transpose
a.'	transpose(a)	Matrix transpose
command window output	display(a)	Display method

Operation	M-File	Description
`[a b]`	`horzcat(a,b,...)`	Horizontal concatenation
`[a; b]`	`vertcat(a,b,...)`	Vertical concatenation
`a(s1,s2,...sn)`	`subsref(a,s)`	Subscripted reference
`a(s1,...,sn) = b`	`subsasgn(a,s,b)`	Subscripted assignment
`b(a)`	`subsindex(a)`	Subscript index

Overloading Functions

You can overload any function by creating a function of the same name in the class directory. When a function is invoked on an object, MATLAB always looks in the class directory before any other location on the search path. To overload the `plot` function for a class of objects, for example, simply place your version of `plot.m` in the appropriate class directory.

Examples of Overloaded Functions

See the following sections for examples of overloaded functions:

- "Overloading Functions for the Polynom Class" on page 9-34
- "The Portfolio pie3 Method" on page 9-61

Example — A Polynomial Class

In this section...

"Polynom Data Structure" on page 9-26

"Polynom Methods" on page 9-26

"The Polynom Constructor Method" on page 9-27

"Converter Methods for the Polynom Class" on page 9-28

"The Polynom display Method" on page 9-30

"The Polynom subsref Method" on page 9-31

"Overloading Arithmetic Operators for polynom" on page 9-32

"Overloading Functions for the Polynom Class" on page 9-34

"Listing Class Methods" on page 9-36

Polynom Data Structure

This example implements a MATLAB data type for polynomials by defining a new class called `polynom`. The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in decreasing order. Therefore, a polynom object `p` is a structure with a single field, `p.c`, containing the coefficients. This field is accessible only within the methods in the `@polynom` directory.

Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method `polynom.m`
- A polynom to double converter
- A polynom to char converter
- A `display` method
- A `subsref` method

- Overloaded +, -, and * operators
- Overloaded roots, polyval, plot, and diff functions

The Polynom Constructor Method

Here is the polynom class constructor, @polynom/polynom.m.

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
%   p = POLYNOM(v) creates a polynomial object from vector v,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

Constructor Calling Syntax

You can call the polynom constructor method with one of three different arguments:

- No input argument — If you call the constructor function with no arguments, it returns a polynom object with empty fields.
- Input argument is an object — If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The isa function (pronounced “is a”) checks for this situation.
- Input argument is a coefficient vector — If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the .c field of the object’s structure. The class function creates the polynom object, which is then returned by the constructor.

An example use of the polynom constructor is the statement

```
p = polynom([1 0 -2 -5])
```

This creates a polynomial with the specified coefficients.

Converter Methods for the Polynom Class

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are `double` and `char`. Conversion to `double` produces the MATLAB traditional matrix, although this may not be appropriate for some classes. Conversion to `char` is useful for producing printed output.

The Polynom to Double Converter

The double converter method for the polynom class is a very simple M-file, `@polynom/double.m`, which merely retrieves the coefficient vector.

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

On the object `p`,

```
p = polynom([1 0 -2 -5])
```

the statement

```
double(p)
```

returns

```
 ans =
 1 0 -2 -5
```

Having implemented the `double` method, you can use it to call MATLAB functions on polynom objects that require double values as inputs. For example,

```
size(double(p))
 ans =
 1   4
```

The Polynom to Char Converter

The converter to char is a key method because it produces a character string involving the powers of an independent variable, x. Therefore, once you have specified x, the string returned is a syntactically correct MATLAB expression, which you can then evaluate.

Here is @polynom/char.m.

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
   s = '0';
else
   d = length(p.c) - 1;
   s = [];
   for a = p.c;
      if a ~= 0;
         if ~isempty(s)
            if a > 0
               s = [s ' + '];
            else
               s = [s ' - '];
               a = -a;
            end
         end
         if a ~= 1 | d == 0
            s = [s num2str(a)];
            if d > 0
               s = [s '*'];
            end
         end
         if d >= 2
            s = [s 'x^' int2str(d)];
         elseif d == 1
            s = [s 'x'];
         end
      end
      d = d - 1;
   end
```

```
end
```

Evaluating the Output

If you create the polynom object p

```
p = polynom([1 0 -2 -5]);
```

and then call the char method on p

```
char(p)
```

MATLAB produces the result

```
ans =
    x^3 - 2*x - 5
```

The value returned by char is a string that you can pass to eval once you have defined a scalar value for x. For example,

```
x = 3;

eval(char(p))
ans =
    16
```

See "The Polynom subsref Method" on page 9-31 for a better method to evaluate the polynomial.

The Polynom display Method

Here is @polynom/display.m. This method relies on the char method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
disp([inputname(1),' = '])
disp(' ');
```

```
disp(['    ' char(p)])
disp(' ');
```

The statement

```
p = polynom([1 0 -2 -5])
```

creates a polynom object. Since the statement is not terminated with a semicolon, the resulting output is

```
p =
    x^3 - 2*x - 5
```

The Polynom subsref Method

Suppose the design of the polynom class specifies that a subscripted reference to a `polynom` object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object `p`,

```
p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at `x = 3` and `x = 4`.

```
p([3 4])
ans =
    16   51
```

subsref Implementation Details

This implementation takes advantage of the `char` method already defined in the polynom class to produce an expression that can then be evaluated.

```
function b = subsref(a,s)
% SUBSREF
switch s.type
case '()'
   ind = s.subs{:};
   for k = 1:length(ind)
      b(k) = eval(strrep(char(a), 'x', ...
              ['(' num2str(ind(k)) ')']));
   end
otherwise
```

```
    error('Specify value for x as p(x)')
end
```

Once the polynomial expression has been generated by the `char` method, the `strrep` function is used to swap the passed in value for the character `x`. The `eval` function then evaluates the expression and returns the value in the output argument.

Note that if you perform an indexed reference from within other class methods, MATLAB calls the built-in `subsref` or `subsassign`. See “Behavior Within Class Methods” on page 9-15 for more information.

Overloading Arithmetic Operators for polynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, the `plus`, `minus`, and `mtimes` methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

Overloading the + Operator

If either `p` or `q` is a polynom, the expression

```
p + q
```

generates a call to a function `@polynom/plus.m`, if it exists (unless `p` or `q` is an object of a higher precedence, as described in “Object Precedence” on page 9-70).

The following M-file redefines the + operator for the polynom class.

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

```
p + 1
```

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the `polynom` constructor a third time to create the properly typed result.

Overloading the - Operator

You can implement the overloaded minus operator (`-`) using the same approach as the plus (+) operator. MATLAB calls `@polynom/minus.m` to compute `p-q`.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

Overloading the * Operator

MATLAB calls the method `@polynom/mtimes.m` to compute the product `p*q`. The letter `m` at the beginning of the function name comes from the fact that it is overloading the MATLAB *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Using the Overloaded Operators

Given the polynom object

```
p = polynom([1 0 -2 -5])
```

MATLAB calls these two functions `@polynom/plus.m` and `@polynom/mtimes.m` when you issue the statements

```
q = p+1
r = p*q
```

to produce

```
q =
    x^3 - 2*x - 4

r =
    x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading Functions for the Polynom Class

MATLAB already has several functions for working with polynomials represented by coefficient vectors. They should be overloaded to also work with the new polynom object. In many cases, the overloading methods can simply apply the original function to the coefficient field.

Overloading roots for the Polynom Class

The method `@polynom/roots.m` finds the roots of polynom objects.

```
function r = roots(p)
% POLYNOM/ROOTS.  ROOTS(p) is a vector containing the roots of p.
r = roots(p.c);
```

The statement

```
roots(p)
```

results in

```
ans =
    2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

Overloading polyval for the Polynom Class

The function polyval evaluates a polynomial at a given set of points. @polynom/polyval.m uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
   y = y.*x + a;
end
```

Overloading plot for the Polynom Class

The overloaded plot function uses both root and polyval. The function selects the domain of the independent variable to be slightly larger than an interval containing the roots of the polynomial. Then polyval is used to evaluate the polynomial at a few hundred points in the domain.

```
function plot(p)
% POLYNOM/PLOT  PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Overloading diff for the Polynom Class

The method @polynom/diff.m differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Listing Class Methods

The function call

```
methods('classname')
```

or its command form

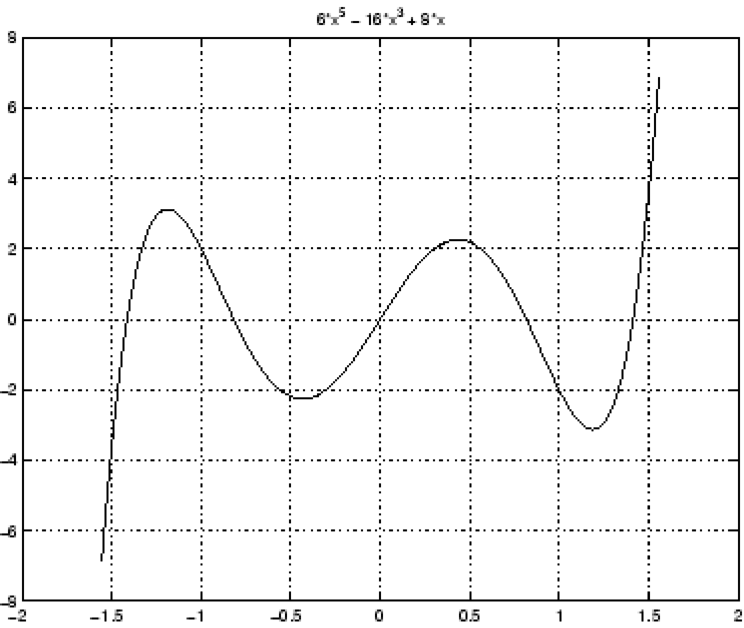
```
methods classname
```

shows all the methods available for a particular class. For the polynom example, the output is

```
methods polynom
Methods for class polynom:

char       display    minus      plot       polynom    roots
diff       double     mtimes     plus       polyval    subsref
```

Plotting the two polynom objects x and p calls most of these methods.

```
x = polynom([1 0]);
p = polynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```



Building on Other Classes

In this section...
"Overview" on page 9-38
"Simple Inheritance" on page 9-38
"Multiple Inheritance" on page 9-40
"Aggregation" on page 9-40

Overview

A MATLAB classes can *inherit* properties and behavior from another MATLAB class. When one object (of the derived class) inherits from another (of the base class), the derived class object includes all the fields of the base class object and can call the base class methods. The base class methods can access those fields that the derived class object inherited from the base class, but not fields new to the derived class.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing derived class objects to take advantage of code that exists for base class objects. Inheritance enables a derived class object to behave exactly like a base class object, which facilitates the development of related classes that behave similarly, but are implemented differently.

There are two kinds of inheritance:

- Simple inheritance, in which a derived class object inherits characteristics from one base class.
- Multiple inheritance, in which a derived class object inherits characteristics from more than one parent class.

This section also discusses a related topic, aggregation. Aggregation allows one object to contain another object as one of its fields.

Simple Inheritance

A class that inherits attributes from a single base class, and adds new attributes of its own, uses simple inheritance. Inheritance implies that objects

belonging to the derived class have the same fields as the base class, as well as any fields added by the derived class.

Base class methods can operate on objects belonging to the derived class. However, derived class methods cannot operate on objects belonging to the base class. You cannot access the base class object's fields directly from the derived class; you must use access methods defined for the base class.

Derived Class Constructor

The constructor function for a derived class has two special characteristics:

- It calls the constructor function for the base class to create the inherited fields.
- It requires a special calling syntax for the `class` function, specifying both the derived class and the base class.

The syntax for establishing simple inheritance using the `class` function is:

```
derivedObj = class(derivedObj, 'derivedClass', baseObj);
```

Simple inheritance can span more than one generation. If a base class is itself an inherited class, the derived class object automatically inherits from the original base class.

Visibility of Class Properties and Methods

The base class does not have access to the derived class properties. The derived class cannot access the base class properties directly, but must use base class access methods (e.g., `get` or `subsref` method) to access the base class properties. From the derived class methods, this access is accomplished via the base class field in the derived class structure. For example, when a constructor creates a derived class object `c`,

```
c = class(c, 'derivedClassname', baseObject);
```

MATLAB automatically creates a field, `c.`*`baseClassname`*, in the object's structure that contains the base object. You could then have a statement in the derived class display method that calls the base class display method.

```
display(c.baseClassname)
```

See “Designing the Stock Class” on page 9-49 for examples that use simple inheritance.

Multiple Inheritance

In the multiple inheritance case, a class of objects inherits attributes from more than one base class. The derived class object gets fields from all the base classes, as well as fields of its own.

Multiple inheritance can encompass more than one generation. For example, each of the base class objects could have inherited fields from multiple base class objects, and so on. Multiple inheritance is implemented in the constructors by calling `class` with more than three arguments.

```
obj = class(structure,'classname',baseclass1,baseclass2,...)
```

You can append as many base class arguments as desired to the class input list.

Nonunique Method Names in Base Classes

Multiple base classes can have associated methods of the same name. In this case, MATLAB calls the method associated with the base class that appears first in the `class` function call in the constructor function. There is no way to access subsequent base class functions of this name.

Aggregation

In addition to standard inheritance, MATLAB objects support containment or *aggregation*. That is, one object can contain (embed) another object as one of its fields. For example, a rational object might use two polynom objects, one for the numerator and one for the denominator.

You can call a method for the contained object only from within a method for the outer object. When determining which version of a function to call, MATLAB considers only the outermost containing class of the objects passed as arguments; the classes of any contained objects are ignored.

See “Example — The Portfolio Container” on page 9-58 for an example of aggregation.

Example — Assets and Asset Subclasses

In this section...

"Inheritance Model for the Asset Class" on page 9-41

"Asset Class Design" on page 9-42

"Other Asset Methods" on page 9-43

"The Asset Constructor Method" on page 9-43

"The Asset get Method" on page 9-44

"The Asset set Method" on page 9-45

"The Asset subsref Method" on page 9-46

"The Asset subsasgn Method" on page 9-47

"The Asset display Method" on page 9-48

"The Asset fieldcount Method" on page 9-49

"Designing the Stock Class" on page 9-49

"The Stock Constructor Method" on page 9-50

"The Stock get Method" on page 9-52

"The Stock set Method" on page 9-53

"The Stock subsref Method" on page 9-54

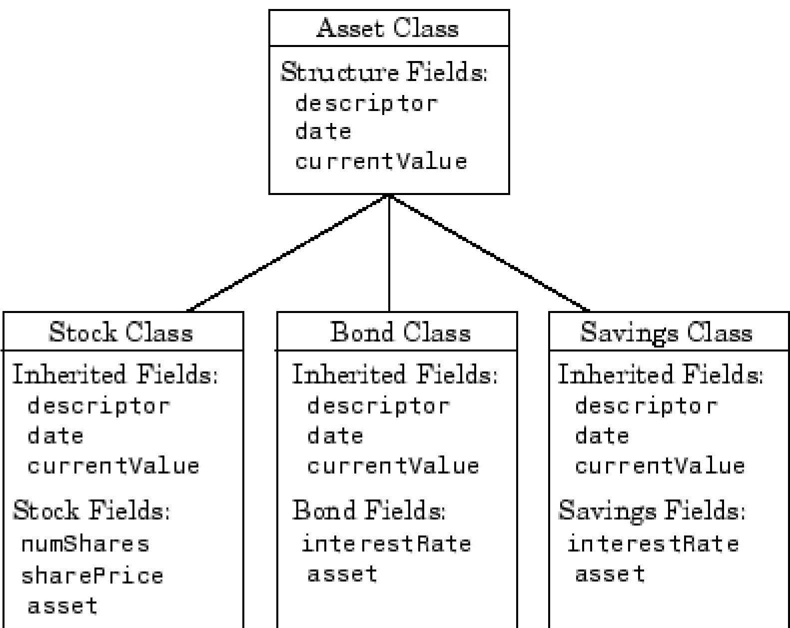
"The Stock subsasgn Method" on page 9-55

"The Stock display Method" on page 9-57

Inheritance Model for the Asset Class

As an example of simple inheritance, consider a general `asset` class that can be used to represent any item that has monetary value. Some examples of an asset are: stocks, bonds, savings accounts, and any other piece of property. In designing this collection of classes, the `asset` class holds the data that is common to all of the specialized asset subclasses. The individual asset subclasses, such as the `stock` class, inherit the `asset` properties and contribute additional properties. The subclasses are "kinds of" assets.

An example of a simple inheritance relationship using an asset base class is shown in this diagram.



As shown in the diagram, the `stock`, `bond`, and `savings` classes inherit structure fields from the `asset` class. In this example, the `asset` class is used to provide storage for data common to all subclasses and to share asset methods with these subclasses. This example shows how to implement the asset and stock classes. The `bond` and `savings` classes can be implemented in a way that is very similar to the `stock` class, as would other types of asset subclasses.

Asset Class Design

The asset class provides storage and access for information common to all asset children. It is not intended to be instantiated directly, so it does not require an extensive set of methods. To serve its purpose, the class needs to contain the following methods:

- Constructor

- get and set
- subsref and subsasgn
- display

Other Asset Methods

The asset class provides inherited data storage for its child classes, but is not instanced directly. The set, get, and display methods provide access to the stored data. It is not necessary to implement the full complement of methods for asset objects (such as converters, end, and subsindex) since only the child classes access the data.

The Asset Constructor Method

The asset class is based on a structure array with four fields:

- descriptor — Identifier of the particular asset (e.g., stock name, savings account number, etc.)
- date — The date the object was created (calculated by the date command)
- type — The type of asset (e.g., savings, bond, stock)
- currentValue — The current value of the asset (calculated from subclass data)

This information is common to asset child objects (stock, bond, and savings), so it is handled from the parent object to avoid having to define the same fields in each child class. This is particularly helpful as the number of child classes increases.

```
function a = asset(varargin)
% ASSET Constructor function for asset object
% a = asset(descriptor, type, currentValue)
switch nargin
case 0
% if no input arguments, create a default object
   a.descriptor = 'none';
   a.date = date;
   a.type = 'none';
   a.currentValue = 0;
```

```
      a = class(a,'asset');
   case 1
   % if single argument of class asset, return it
      if (isa(varargin{1},'asset'))
         a = varargin{1};
      else
         error('Wrong argument type')
      end
   case 3
   % create object using specified values
      a.descriptor = varargin{1};
      a.date = date;
      a.type = varargin{2};
      a.currentValue = varargin{3};
      a = class(a,'asset');
   otherwise
      error('Wrong number of input arguments')
   end
```

The function uses a `switch` statement to accommodate three possible scenarios:

- Called with no arguments, the constructor returns a default asset object.
- Called with one argument that is an asset object, the object is simply returned.
- Called with three arguments (subclass descriptor, type, and current value), the constructor returns a new asset object.

The asset constructor method is not intended to be called directly; it is called from the child constructors since its purpose is to provide storage for common data.

The Asset get Method

The asset class needs methods to access the data contained in asset objects. The following function implements a `get` method for the class. It uses capitalized property names rather than literal field names to provide an interface similar to other MATLAB objects.

```
function val = get(a, propName)
% GET Get asset properties from the specified object
% and return the value
switch propName
case 'Descriptor'
   val = a.descriptor;
case 'Date'
   val = a.date;
case 'CurrentValue'
   val = a.currentValue;
otherwise
   error([propName,' Is not a valid asset property'])
end
```

This function accepts an object and a property name and uses a `switch` statement to determine which field to access. This method is called by the subclass `get` methods when accessing the data in the inherited properties. See "The Stock get Method" on page 9-52 for an example.

The Asset set Method

The asset class `set` method is called by subclass `set` methods. This method accepts an asset object and variable length argument list of property name/property value pairs and returns the modified object.

```
function a = set(a,varargin)
% SET Set asset properties and return the updated object
propertyArgIn = varargin;
while length(propertyArgIn) >= 2,
   prop = propertyArgIn{1};
   val = propertyArgIn{2};
   propertyArgIn = propertyArgIn(3:end);
   switch prop
   case 'Descriptor'
      a.descriptor = val;
   case 'Date'
      a.date = val;
   case 'CurrentValue'
      a.currentValue = val;
   otherwise
```

```
        error('Asset properties: Descriptor, Date, CurrentValue')
    end
end
```

Subclass `set` methods call the asset `set` method and require the capability to return the modified object since MATLAB does not support passing arguments by reference. See "The Stock set Method" on page 9-53 for an example.

The Asset subsref Method

The `subsref` method provides access to the data contained in an asset object using one-based numeric indexing and structure field name indexing. The outer `switch` statement determines if the index is a numeric or field name syntax. The inner `switch` statements map the index to the appropriate value.

MATLAB calls `subsref` whenever you make a subscripted reference to an object (e.g., `A(i)`, `A{i}`, or `A.fieldname`).

```
function b = subsref(a,index)
%SUBSREF Define field name indexing for asset objects
switch index.type
case '()'
    switch index.subs{:}
    case 1
        b = a.descriptor;
    case 2
        b = a.date;
    case 3
        b = a.currentValue;
    otherwise
        error('Index out of range')
    end
case '.'
    switch index.subs
    case 'descriptor'
        b = a.descriptor;
    case 'date'
        b = a.date;
    case 'currentValue'
        b = a.currentValue;
```

```
      otherwise
         error('Invalid field name')
      end
   case '{}'
      error('Cell array indexing not supported by asset objects')
   end
```

See the "The Stock subsref Method" on page 9-54 for an example of how the child subsref method calls the parent subsref method.

The Asset subsasgn Method

The subsasgn method is the assignment equivalent of the subsref method. This version enables you to change the data contained in an object using one-based numeric indexing and structure field name indexing. The outer switch statement determines if the index is a numeric or field name syntax. The inner switch statements map the index value to the appropriate value in the stock structure.

MATLAB calls subsasgn whenever you execute an assignment statement (e.g., A(i) = val, A{i} = val, or A.fieldname = val).

```
function a = subsasgn(a,index,val)
% SUBSASGN Define index assignment for asset objects
switch index.type
case '()'
   switch index.subs{:}
   case 1
      a.descriptor = val;
   case 2
      a.date = val;
   case 3
      a.currentValue = val;
   otherwise
      error('Index out of
   end
case '.'
   switch index.subs
   case 'descriptor'
      a.descriptor = val;
```

```
      case 'date'
         a.date = val;
      case 'currentValue'
         a.currentValue = val;
      otherwise
         error('Invalid field name')
      end
   end
```

The subsasgn method enables you to assign values to the asset object data structure using two techniques. For example, suppose you have a child stock object s. (If you want to run this statement, you first need to create a stock constructor method.)

```
s = stock('XYZ',100,25);
```

Within stock class methods, you could change the descriptor field with either of the following statements

```
s.asset(1) = 'ABC';
```

or

```
s.asset.descriptor = 'ABC';
```

See the "The Stock subsasgn Method" on page 9-55 for an example of how the child subsasgn method calls the parent subsasgn method.

The Asset display Method

The asset display method is designed to be called from child-class display methods. Its purpose is to display the data it stores for the child object. The method simply formats the data for display in a way that is consistent with the formatting of the child's display method.

```
function display(a)
% DISPLAY(a) Display an asset object
stg = sprintf(...
   'Descriptor: %s\nDate: %s\nType: %s\nCurrent Value:%9.2f',...
   a.descriptor,a.date,a.type,a.currentValue);
disp(stg)
```

The stock class display method can now call this method to display the data stored in the parent class. This approach isolates the stock `display` method from changes to the asset class. See "The Stock display Method" on page 9-57 for an example of how this method is called.

The Asset fieldcount Method

The asset `fieldcount` method returns the number of fields in the asset object data structure. `fieldcount` enables asset child methods to determine the number of fields in the asset object during execution, rather than requiring the child methods to have knowledge of the asset class. This allows you to make changes to the number of fields in the asset class data structure without having to change child-class methods.

```
function numFields = fieldcount(assetObj)
% Determines the number of fields in an asset object
% Used by asset child class methods
numFields = length(fieldnames(assetObj));
```

The `struct` function converts an object to its equivalent data structure, enabling access to the structure's contents.

Designing the Stock Class

A stock object is designed to represent one particular asset in a person's investment portfolio. This object contains two properties of its own and inherits three properties from its parent asset object.

Stock properties:

- `NumberShares` — The number of shares for the particular stock object.
- `SharePrice` — The value of each share.

Asset properties:

- `Descriptor` — The identifier of the particular asset (e.g., stock name, savings account number, etc.).
- `Date` — The date the object was created (calculated by the `date` command).
- `CurrentValue` — The current value of the asset.

Note that the property names are not actually the same as the field names of the structure array used internally by stock and asset objects. The property name interface is controlled by the stock and asset `set` and `get` methods and is designed to resemble the interface of other MATLAB object properties.

The `asset` field in the stock object structure contains the parent asset object and is used to access the inherited fields in the parent structure.

Stock Class Methods

The stock class implements the following methods:

- Constructor
- `get` and `set`
- `subsref` and `subsasgn`
- `display`

The Stock Constructor Method

The stock constructor creates a stock object from three input arguments:

- The stock name
- The number of shares
- The share price

The constructor must create an asset object from within the stock constructor to be able to specify it as a parent to the stock object. The stock constructor must, therefore, call the asset constructor. The `class` function, which is called to create the stock object, defines the asset object as the parent.

Keep in mind that the asset object is created in the temporary workspace of the stock constructor function and is stored as a field (`.asset`) in the stock structure. The stock object inherits the asset fields, but the asset object is not returned to the base workspace.

```
function s = stock(varargin)
% STOCK Stock class constructor.
% s = stock(descriptor, numShares, sharePrice)
```

```
switch nargin
case 0
   % if no input arguments, create a default object
   s.numShares = 0;
   s.sharePrice = 0;
   a = asset('none',0);
   s = class(s,'stock',a);
case 1
   % if single argument of class stock, return it
   if (isa(varargin{1},'stock'))
      s = varargin{1};
   else
      error('Input argument is not a stock object')
   end
case 3
   % create object using specified values
   s.numShares = varargin{2};
   s.sharePrice = varargin{3};
   a = asset(varargin{1},'stock',varargin{2} * varargin{3});
   s = class(s,'stock',a);
otherwise
   error('Wrong number of input arguments')
end
```

Constructor Calling Syntax

The stock constructor method can be called in one of three ways:

- No input argument — If called with no arguments, the constructor returns a default object with empty fields.
- Input argument is a stock object — If called with a single input argument that is a stock object, the constructor returns the input argument. A single argument that is not a stock object generates an error.
- Three input arguments — If there are three input arguments, the constructor uses them to define the stock object.

Otherwise, if none of the above three conditions are met, return an error.

For example, this statement creates a stock object to record the ownership of 100 shares of XYZ corporation stocks with a price per share of 25 dollars.

```
XYZStock = stock('XYZ',100,25);
```

The Stock get Method

The get method provides a way to access the data in the stock object using a "property name" style interface, similar to Handle Graphics. While in this example the property names are similar to the structure field name, they can be quite different. You could also choose to exclude certain fields from access via the get method or return the data from the same field for a variety of property names, if such behavior suits your design.

```
function val = get(s,propName)
% GET Get stock property from the specified object
% and return the value. Property names are: NumberShares
% SharePrice, Descriptor, Date, CurrentValue
switch propName
case 'NumberShares'
   val = s.numShares;
case 'SharePrice'
   val = s.sharePrice;
case 'Descriptor'
   val = get(s.asset,'Descriptor'); % call asset get method
case 'Date'
   val = get(s.asset,'Date');
case 'CurrentValue'
   val = get(s.asset,'CurrentValue');
otherwise
   error([propName ,'Is not a valid stock property'])
end
```

Note that the asset object is accessed via the stock object's asset field (s.asset). MATLAB automatically creates this field when the class function is called with the parent argument.

The Stock set Method

The `set` method provides a "property name" interface like the `get` method. It is designed to update the number of shares, the share value, and the descriptor. The current value and the date are automatically updated.

```
function s = set(s,varargin)
% SET Set stock properties to the specified values
% and return the updated object
propertyArgIn = varargin;
while length(propertyArgIn) >= 2,
   prop = propertyArgIn{1};
   val = propertyArgIn{2};
   propertyArgIn = propertyArgIn(3:end);
   switch prop
   case 'NumberShares'
      s.numShares = val;
   case 'SharePrice'
      s.sharePrice = val;
   case 'Descriptor'
      s.asset = set(s.asset,'Descriptor',val);
   otherwise
      error('Invalid property')
   end
end
s.asset = set(s.asset,'CurrentValue',...
   s.numShares * s.sharePrice,'Date',date);
```

Note that this function creates and returns a new stock object with the new values, which you then copy over the old value. For example, given the stock object,

```
s = stock('XYZ',100,25);
```

the following `set` command updates the share price.

```
s = set(s,'SharePrice',36);
```

It is necessary to copy over the original stock object (i.e., assign the output to `s`) because MATLAB does not support passing arguments by reference. Hence the `set` method actually operates on a copy of the object.

The Stock subsref Method

The subsref method defines subscripted indexing for the stock class. In this example, subsref is implemented to enable numeric and structure field name indexing of stock objects.

```
function b = subsref(s,index)
% SUBSREF Define field name indexing for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
   if (index.subs{:} <= fc)
      b = subsref(s.asset,index);
   else
      switch index.subs{:} - fc
      case 1
         b = s.numShares;
      case 2
         b = s.sharePrice;
      otherwise
         error(['Index must be in the range 1 to ', ...
            num2str(fc + 2)])
      end
   end
case '.'
   switch index.subs
   case 'numShares'
      b = s.numShares;
   case 'sharePrice'
      b = s.sharePrice;
   otherwise
      b = subsref(s.asset,index);
   end
end
```

The outer switch statement determines if the index is a numeric or field name syntax.

The fieldcount asset method determines how many fields there are in the asset structure, and the if statement calls the asset subsref method for

indices 1 to fieldcount. See “The Asset fieldcount Method” on page 9-49 and “The Asset subsref Method” on page 9-46 for a description of these methods.

Numeric indices greater than the number returned by fieldcount are handled by the inner switch statement, which maps the index value to the appropriate field in the stock structure.

Field name indexing assumes field names other than numShares and sharePrice are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset subsref method performs field-name error checking.

See the subsref help entry for general information on implementing this method.

The Stock subsasgn Method

The subsasgn method enables you to change the data contained in a stock object using numeric indexing and structure field name indexing. MATLAB calls subsasgn whenever you execute an assignment statement (e.g., A(i) = val, A{i} = val, or A.fieldname = val).

```
function s = subsasgn(s,index,val)
% SUBSASGN Define index assignment for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
   if (index.subs{:} <= fc)
      s.asset = subsasgn(s.asset,index,val);
   else
      switch index.subs{:}-fc
      case 1
         s.numShares = val;
      case 2
         s.sharePrice = val;
      otherwise
         error(['Index must be in the range 1 to ', ...
            num2str(fc + 2)])
      end
   end
```

```
case '.'
   switch index.subs
   case 'numShares'
      s.numShares = val;
   case 'sharePrice'
      s.sharePrice = val;
   otherwise
      s.asset = subsasgn(s.asset,index,val);
   end
end
```

The outer `switch` statement determines if the index is a numeric or field name syntax.

The `fieldcount` asset method determines how many fields there are in the asset structure and the `if` statement calls the asset `subsasgn` method for indices `1` to `fieldcount`. See "The Asset fieldcount Method" on page 9-49 and "The Asset subsasgn Method" on page 9-47 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner `switch` statement, which maps the index value to the appropriate field in the stock structure.

Field name indexing assumes field names other than `numShares` and `sharePrice` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset `subsasgn` method performs field-name error checking.

The `subsasgn` method enables you to assign values to stock object data structure using two techniques. For example, suppose you have a stock object

```
s = stock('XYZ',100,25)
```

You could change the `descriptor` field with either of the following statements

```
s(1) = 'ABC';
```

or

```
s.descriptor = 'ABC';
```

See the subsasgn help entry for general information on assignment statements in MATLAB.

The Stock display Method

When you issue the statement (without terminating with a semicolon)

```
XYZStock = stock('XYZ',100,25)
```

MATLAB looks for a method in the @stock directory called display. The display method for the stock class produces this output.

```
Descriptor: XYZ
Date: 17-Nov-1998
Type: stock
Current Value:  2500.00
Number of shares: 100
Share price: 25.00
```

Here is the stock display method.

```
function display(s)
% DISPLAY(s) Display a stock object
display(s.asset)
stg = sprintf('Number of shares: %g\nShare price: %3.2f\n',...
   s.numShares,s.sharePrice);
disp(stg)
```

First, the parent asset object is passed to the asset display method to display its fields (MATLAB calls the asset display method because the input argument is an asset object). The stock object's fields are displayed in a similar way using a formatted text string.

Note that if you did not implement a stock class display method, MATLAB would call the asset display method. This would work, but would display only the descriptor, date, type, and current value.

Example — The Portfolio Container

In this section...

"Overview" on page 9-58
"Designing the Portfolio Class" on page 9-58
"The Portfolio Constructor Method" on page 9-59
"The Portfolio display Method" on page 9-61
"The Portfolio pie3 Method" on page 9-61
"Creating a Portfolio" on page 9-62

Overview

Aggregation is the containment of one class by another class. The basic relationship is: each contained class "is a part of" the container class.

For example, consider a financial portfolio class as a container for a set of assets (stocks, bonds, savings, etc.). Once the individual assets are grouped, they can be analyzed, and useful information can be returned. The contained objects are not accessible directly, but only via the portfolio class methods.

See "Example — Assets and Asset Subclasses" on page 9-41 for information about the assets collected by this portfolio class.

Designing the Portfolio Class

The portfolio class is designed to contain the various assets owned by a given individual and provide information about the status of his or her investment portfolio. This example implements a somewhat over-simplified portfolio class that

- Contains an individual's assets
- Displays information about the portfolio contents
- Displays a 3-D pie chart showing the relative mix of asset types in the portfolio

Required Portfolio Methods

The portfolio class implements only three methods:

- `portfolio` — The portfolio constructor.
- `display` — Displays information about the portfolio contents.
- `pie3` — Overloaded version of `pie3` function designed to take a single portfolio object as an argument.

Since a portfolio object contains other objects, the portfolio class methods can use the methods of the contained objects. For example, the portfolio `display` method calls the stock class `display` method, and so on.

The Portfolio Constructor Method

The portfolio constructor method takes as input arguments a client's name and a variable length list of asset subclass objects (stock, bond, and savings objects in this example). The portfolio object uses a structure array with the following fields:

- `name` — The client's name.
- `indAssets` — The array of asset subclass objects (stock, bond, savings).
- `totalValue` — The total value of all assets. The constructor calculates this value from the objects passed in as arguments.
- `accountNumber` — The account number. This field is assigned a value only when you save a portfolio object (see "Saving and Loading Objects" on page 9-64).

```
function p = portfolio(name,varargin)
% PORTFOLIO Create a portfolio object containing the
% client's name and a list of assets
switch nargin
case 0
   % if no input arguments, create a default object
   p.name = 'none';
   p.totalValue = 0;
   p.indAssets = {};
```

```
      p.accountNumber = '';
      p = class(p,'portfolio');
   case 1
      % if single argument of class portfolio, return it
      if isa(name,'portfolio')
         p = name;
      else
         disp([inputname(1) ' is not a portfolio object'])
         return
      end
   otherwise
      % create object using specified arguments
      p.name = name;
      p.totalValue = 0;
      for k = 1:length(varargin)
         p.indAssets(k) = {varargin{k}};
         assetValue = get(p.indAssets{k},'CurrentValue');
         p.totalValue = p.totalValue + assetValue;
      end
      p.accountNumber = '';
      p = class(p,'portfolio');
   end
```

Constructor Calling Syntax

The portfolio constructor method can be called in one of three different ways:

- No input arguments — If called with no arguments, it returns an object with empty fields.
- Input argument is an object — If the input argument is already a portfolio object, MATLAB returns the input argument. The `isa` function checks for this case.
- More than two input arguments — If there are more than two input arguments, the constructor assumes the first is the client's name and the rest are asset subclass objects. A more thorough implementation would perform more careful input argument checking, for example, using the `isa` function to determine if the arguments are the correct class of objects.

The Portfolio display Method

The portfolio `display` method lists the contents of each contained object by calling the object's `display` method. It then lists the client name and total asset value.

```
function display(p)
% DISPLAY Display a portfolio object
for k = 1:length(p.indAssets)
   display(p.indAssets{k})
end
stg = sprintf('\nAssets for Client: %s\nTotal Value: %9.2f\n',...
p.name,p.totalValue);
disp(stg)
```

The Portfolio pie3 Method

The portfolio class overloads the MATLAB `pie3` function to accept a portfolio object and display a 3-D pie chart illustrating the relative asset mix of the client's portfolio. MATLAB calls the `@portfolio/pie3.m` version of `pie3` whenever the input argument is a single portfolio object.

```
function pie3(p)
% PIE3 Create a 3-D pie chart of a portfolio
stockAmt = 0; bondAmt = 0; savingsAmt = 0;
for k = 1:length(p.indAssets)
   if isa(p.indAssets{k}, 'stock')
      stockAmt = stockAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   elseif isa(p.indAssets{k}, 'bond')
      bondAmt = bondAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   elseif isa(p.indAssets{k}, 'savings')
      savingsAmt = savingsAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   end
end
i = 1;
if stockAmt ~= 0
   label(i) = {'Stocks'};
   pieVector(i) = stockAmt;
   i = i + 1;
```

```
end
if bondAmt ~= 0
   label(i) = {'Bonds'};
   pieVector(i) = bondAmt;
   i = i + 1;
end
if savingsAmt ~= 0
   label(i) = {'Savings'};
   pieVector(i) = savingsAmt;
end
pie3(pieVector, label)
set(gcf, 'Renderer', 'zbuffer')
set(findobj(gca, 'Type', 'Text'), 'FontSize', 14)
cm = gray(64);
colormap(cm(48:end, :))
stg(1) = {['Portfolio Composition for ', p.name]};
stg(2) = {['Total Value of Assets: $', num2str(p.totalValue)]};
title(stg, 'FontSize', 12)
```

There are three parts in the overloaded `pie3` method.

- The first uses the asset subclass `get` methods to access the `CurrentValue` property of each contained object. The total value of each class is summed.
- The second part creates the pie chart labels and builds a vector of graph data, depending on which objects are present.
- The third part calls the MATLAB `pie3` function, makes some font and colormap adjustments, and adds a title.

Creating a Portfolio

Suppose you have implemented a collection of asset subclasses in a manner similar to the stock class. You can then use a portfolio object to present the individual's financial portfolio. For example, given the following assets

```
XYZStock = stock('XYZ', 200, 12);
SaveAccount = savings('Acc # 1234', 2000, 3.2);
Bonds = bond('U.S. Treasury', 1600, 12);
```

create a portfolio object:

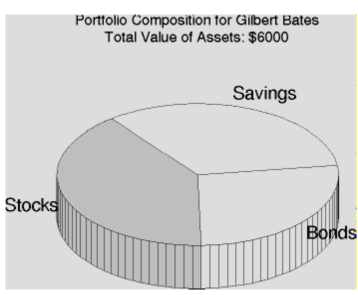
```
p = portfolio('Gilbert Bates',XYZStock,SaveAccount,Bonds)
```

The portfolio display method summarizes the portfolio contents (because this statement is not terminated by a semicolon).

```
Descriptor: XYZ
Date: 24-Nov-1998
Current Value:   2400.00
Type: stock
Number of shares: 200
Share price: 12.00
Descriptor: Acc # 1234
Date: 24-Nov-1998
Current Value:   2000.00
Type: savings
Interest Rate: 3.2%
Descriptor: U.S. Treasury
Date: 24-Nov-1998
Current Value:   1600.00
Type: bond
Interest Rate: 12%
Assets for Client: Gilbert Bates
Total Value:   6000.00
```

The portfolio pie3 method displays the relative mix of assets using a pie chart.

```
pie3(p)
```



Saving and Loading Objects

You can use the MATLAB `save` and `load` commands to save and retrieve user-defined objects to and from `.mat` files, just like any other variables.

When you load objects, MATLAB calls the object's class constructor to register the object in the workspace. The constructor function for the object class you are loading must be able to be called with no input arguments and return a default object. See "Guidelines for Writing a Constructor" on page 9-10 for more information.

When you issue a `save` or `load` command on objects, MATLAB looks for class methods called `saveobj` and `loadobj` in the class directory. You can overload these methods to modify the object before the save or load operation. For example, you could define a `saveobj` method that saves related data along with the object or you could write a `loadobj` method that updates objects to a newer version when this type of object is loaded into the MATLAB workspace.

Example — Defining saveobj and loadobj for Portfolio

In this section...

"Methods Executed by Save and Load" on page 9-65

"Summary of Code Changes" on page 9-65

"The saveobj Method" on page 9-66

"The loadobj Method" on page 9-66

"Changing the Portfolio Constructor" on page 9-67

"The Portfolio subsref Method" on page 9-68

Methods Executed by Save and Load

In the section "Example — The Portfolio Container" on page 9-58, portfolio objects are used to collect information about a client's investment portfolio. Suppose you decide to add an account number to each portfolio object that is saved. You can define a portfolio `saveobj` method to carry out this task automatically during the save operation.

Suppose further that you have already saved a number of portfolio objects without the account number. You want to update these objects during the load operation so that they are still valid portfolio objects. You can do this by defining a `loadobj` method for the portfolio class.

Summary of Code Changes

To implement the account number scenario, you need to add or change the following functions:

- `portfolio` — The portfolio constructor method needs to be modified to create a new field, `accountNumber`, which is initialized to the empty string when an object is created.
- `saveobj` — A new portfolio method designed to add an account number to a portfolio object during the save operation, only if the object does not already have one.

- `loadobj` — A new portfolio method designed to update older versions of portfolio objects that were saved before the account number structure field was added.
- `subsref` — A new portfolio method that enables subscripted reference to portfolio objects outside of a portfolio method.
- `getAccountNumber` — a MATLAB function that returns an account number that consists of the first three letters of the client's name.

New Portfolio Class Behavior

With the additions and changes made in this example, the portfolio class now

- Includes a field for an account number
- Adds the account number when a portfolio object is saved for the first time
- Automatically updates the older version of portfolio objects when you load them into the MATLAB workspace

The saveobj Method

MATLAB looks for the portfolio `saveobj` method whenever the `save` command is passed a portfolio object. If `@portfolio/saveobj` exists, MATLAB passes the portfolio object to `saveobj`, which must then return the modified object as an output argument. The following implementation of `saveobj` determines if the object has already been assigned an account number from a previous save operation. If not, `saveobj` calls `getAccountNumber` to obtain the number and assigns it to the `accountNumber` field.

```
function b = saveobj(a)
if isempty(a.accountNumber)
   a.accountNumber = getAccountNumber(a);
end
b = a;
```

The loadobj Method

MATLAB looks for the portfolio `loadobj` method whenever the `load` command detects portfolio objects in the `.mat` file being loaded. If `loadobj` exists, MATLAB passes the portfolio object to `loadobj`, which must then return the modified object as an output argument. The output argument is then loaded into the workspace.

If the input object does not match the current definition as specified by the constructor function, then MATLAB converts it to a structure containing the same fields and the object's structure with all the values intact (that is, you now have a structure, not an object).

The following implementation of `loadobj` first uses `isa` to determine whether the input argument is a portfolio object or a structure. If the input is an object, it is simply returned since no modifications are necessary. If the input argument has been converted to a structure by MATLAB, then the new `accountNumber` field is added to the structure and is used to create an updated portfolio object.

```
function b = loadobj(a)
% loadobj for portfolio class
if isa(a,'portfolio')
   b = a;
else % a is an old version
   a.accountNumber = getAccountNumber(a);
   b = class(a,'portfolio');
end
```

Changing the Portfolio Constructor

The portfolio structure array needs an additional field to accommodate the account number. To create this field, add the line

```
p.accountNumber = '';
```

to `@portfolio/portfolio.m` in both the zero argument and variable argument sections.

The getAccountNumber Function

In this example, `getAccountNumber` is a MATLAB function that returns an account number composed of the first three letters of the client name prepended to a series of digits. To illustrate implementation techniques, `getAccountNumber` is not a portfolio method so it cannot access the portfolio object data directly. Therefore, it is necessary to define a portfolio `subsref` method that enables access to the name field in a portfolio object's structure.

For this example, getAccountNumber simply generates a random number, which is formatted and concatenated with elements 1 to 3 from the portfolio name field.

```
function n = getAccountNumber(p)
% provides a account number for object p
n = [upper(p.name(1:3)) strcat(num2str(round(rand(1,7)*10))')'];
```

Note that the portfolio object is indexed by field name, and then by numerical subscript to extract the first three letters. The subsref method must be written to support this form of subscripted reference.

The Portfolio subsref Method

When MATLAB encounters a subscripted reference, such as that made in the getAccountNumber function

```
p.name(1:3)
```

MATLAB calls the portfolio subsref method to interpret the reference. If you do not define a subsref method, the above statement is undefined for portfolio objects (recall that here p is an object, not just a structure).

The portfolio subsref method must support field-name and numeric indexing for the getAccountNumber function to access the portfolio name field.

```
function b = subsref(p,index)
% SUBSREF Define field name indexing for portfolio objects
switch index(1).type
case '.'
   switch index(1).subs
   case 'name'
      if length(index)== 1
         b = p.name;
      else
         switch index(2).type
         case '()'
            b = p.name(index(2).subs{:});
         end
      end
   end
```

```
end
```

Note that the portfolio implementation of `subsref` is designed to provide access to specific elements of the `name` field; it is not a general implementation that provides access to all structure data, such as the stock class implementation of `subsref`.

See the `subsref` help entry for more information about indexing and objects.

Object Precedence

In this section...

"How MATLAB Determines Precedence" on page 9-70
"Specifying Precedence of User-Defined Classes" on page 9-71

How MATLAB Determines Precedence

Object precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

```
objectA + objectB
```

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the `inferiorto` and `superiorto` functions.

For example, in the section "Example — A Polynomial Class" on page 9-26 the polynom class defines a `plus` method that enables addition of polynom objects. Given the polynom object `p`

```
p = polynom([1 0 -2 -5])
p =
    x^3-2*x-5
```

The expression,

```
1 + p
ans =
    x^3-2*x-4
```

calls the polynom `plus` method (which converts the double, 1, to a polynom object, and then adds it to `p`). The user-defined polynom class has precedence over the MATLAB double class.

Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferiorto` or `superiorto` function in the class constructor.

The `inferiorto` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferiorto` function is

```
inferiorto('class1','class2',...)
```

You can specify multiple classes in the argument list, placing the class below many other classes in the hierarchy.

Similarly, the `superiorto` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superiorto` function is

```
superiorto('class1','class2',...)
```

Location in the Hierarchy

If *objectA* is above *objectB* in the precedence hierarchy, then the expression

```
objectA + objectB
```

calls `@`*classA*`/plus.m`. Conversely, if *objectB* is above *objectA* in the precedence hierarchy, then MATLAB calls `@`*classB*`/plus.m`.

See "How MATLAB Determines Which Method to Call" on page 9-72 for related information.

How MATLAB Determines Which Method to Call

In this section...

"Overview" on page 9-72

"Selecting a Method" on page 9-72

"Querying Which Method MATLAB Will Call" on page 9-75

Overview

In MATLAB, functions exist in directories in the computer's file system. A directory may contain many functions (M-files). Function names are unique only within a single directory (e.g., more than one directory may contain a function called `pie3`). When you type a function name on the command line, MATLAB must search all the directories it is aware of to determine which function to call. This list of directories is called the *MATLAB path*.

When looking for a function, MATLAB searches the directories in the order they are listed in the path, and calls the first function whose name matches the name of the specified function.

If you write an M-file called `pie3.m` and put it in a directory that is searched before the `specgraph` directory that contains the MATLAB `pie3` function, then MATLAB uses your `pie3` function instead.

Object-oriented programming allows you to have many methods (MATLAB functions located in class directories) with the same name and enables MATLAB to determine which method to use based on the type or class of the variables passed to the function. For example, if `p` is a portfolio object, then

```
pie3(p)
```

calls `@portfolio/pie3.m` because the argument is a portfolio object.

Selecting a Method

When you call a method for which there are multiple versions with the same name, MATLAB determines the method to call by:

- Looking at the classes of the objects in the argument list to determine which argument has the highest object precedence; the class of this object controls the method selection and is called the *dispatch type*.
- Applying the *function precedence order* to determine which of possibly several implementations of a method to call. This order is determined by the location and type of function.

Determining the Dispatch Type

MATLAB first determines which argument controls the method selection. The class type of this argument then determines the class in which MATLAB searches for the method. The controlling argument is either

- The argument with the highest precedence, or
- The leftmost of arguments having equal precedence

User-defined objects take precedence over the MATLAB built-in classes such as `double` or `char`. You can set the relative precedence of user-defined objects with the `inferiorto` and `superiorto` functions, as described in "Object Precedence" on page 9-70.

MATLAB searches for functions by name. When you call a function, MATLAB knows the name, number of arguments, and the type of each argument. MATLAB uses the dispatch type to choose among multiple functions of the same name, but does not consider the number of arguments.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules, in the order given:

1. Subfunctions

   Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

2. Private functions

   Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

3. Class constructor functions

   Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

4. Overloaded methods

   MATLAB calls an overloaded method if it is not masked by a subfunction or private function.

5. Current directory

   A function in the current working directory is selected before one elsewhere on the path.

6. Elsewhere on path

   Finally, a function anywhere else on the path is selected.

Selecting Methods from Multiple Directories

There may be a number of directories on the path that contain methods with the same name. MATLAB stops searching when it finds the first implementation of the method on the path, regardless of the implementation type (MEX-file, P-code, M-file).

Selecting Methods from Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is

1. Built-in file
2. MEX-files
3. MDL (Simulink model) file
4. P-code file
5. M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Method MATLAB Will Call

You can determine which method MATLAB will call using the `which` command. For example,

```
which pie3
your_matlab_path/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m      % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

Scheduling Program Execution with Timers

Using a MATLAB Timer Object (p. 10-2)	Step-by-step procedure for using a timer object with a simple example
Creating Timer Objects (p. 10-5)	Using the `timer` function to create a timer object
Working with Timer Object Properties (p. 10-7)	Setting timer object properties and retrieving the values of timer object properties
Starting and Stopping Timers (p. 10-10)	Using the `start` or `startat` function to start timer objects; using the `stop` function to stop them, and blocking the command line
Creating and Executing Callback Functions (p. 10-14)	Creating a callback function and specifying it as the value of a timer object callback property
Timer Object Execution Modes (p. 10-19)	Using the `ExecutionMode` property to control when a timer object executes
Deleting Timer Objects from Memory (p. 10-23)	Using the `delete` function to delete a timer object
Finding Timer Objects in Memory (p. 10-24)	Using the `timerfind` and `timerfindall` functions to determine if timer objects exist in memory

Using a MATLAB Timer Object

In this section...

"Overview" on page 10-2

"Example: Displaying a Message" on page 10-3

Overview

MATLAB includes a timer object that you can use to schedule the execution of MATLAB commands. This section describes how you can create timer objects, start a timer running, and specify the processing that you want performed when a timer fires. A timer is said to *fire* when the amount of time specified by the timer object elapses and the timer object executes the commands you specify.

To use a timer, perform these steps:

1 Create a timer object.

   You use the `timer` function to create a timer object. See "Creating Timer Objects" on page 10-5 for more information.

2 Specify which MATLAB commands you want executed when the timer fires and control other aspects of timer object behavior.

   You use timer object properties to specify this information. To learn about all the properties supported by the timer object, see "Working with Timer Object Properties" on page 10-7. (You can also set timer object properties when you create them, in step 1.)

3 Start the timer object.

   After you create the timer object, you must start it, using either the `start` or `startat` function. See "Starting and Stopping Timers" on page 10-10 for more information.

4 Delete the timer object when you are done with it.

After you are finished using a timer object, you should delete it from memory. See "Deleting Timer Objects from Memory" on page 10-23 for more information.

Note The specified execution time and the actual execution of a timer can vary because timer objects work in the MATLAB single-threaded execution environment. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

Example: Displaying a Message

The following example sets up a timer object that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, `TimerFcn` and `StartDelay`. `TimerFcn` specifies the timer callback function. This is the MATLAB command string or M-file that you want to execute when the timer fires. In the example, the timer callback function sets the value of the MATLAB workspace variable `stat` and executes the MATLAB `disp` command. The `StartDelay` property specifies how much time elapses before the timer fires.

After creating the timer object, the example uses the `start` function to start the timer object. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
                 'StartDelay',10);
start(t)

stat=true;
while(stat==true)
  disp('.')
  pause(1)
end
```

When you execute this code, it produces this output:

```
.
.
.
.
.
.
.
.
.
Timer!

delete(t) % Always delete timer objects after using them.
```

Creating Timer Objects

In this section...
"Creating the Object" on page 10-5
"Naming the Object" on page 10-6

Creating the Object

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the `timer` function. This creates a valid timer object with default values for most properties. The following shows an example of the default timer object and its summary display:

```
t = timer
Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

MATLAB names the timer object `timer-1`. (See "Naming the Object" on page 10-6 for more information.)

To specify the value of timer object properties after you create it, you can use the `set` function. This example sets the value of the `TimerFcn` property and the `StartDelay` property. For more information about timer object properties, see "Working with Timer Object Properties" on page 10-7.

```
set(t,'TimerFcn','disp(''Hello World!'')','StartDelay',5)
```

You can also set timer object properties when you create the timer object by specifying property name and value pairs as arguments to the `timer` function. The following example sets the same properties at object creation time:

```
t = timer('TimerFcn', 'disp(''Hello World!'')','StartDelay',5);
```

Always delete timer objects when you are done using them. See "Deleting Timer Objects from Memory" on page 10-23 for more information.

Naming the Object

MATLAB assigns a name to each timer object you create. This name has the form `'timer-i'`, where *i* is a number representing the total number of timer objects created this session.

For example, the first time you call the timer function to create a timer object, MATLAB names the object `timer-1`. If you call the timer function again to create another timer object, MATLAB names the object `timer-2`.

MATLAB keeps incrementing the number associated with each timer object it creates, even if you delete the timer objects you already created. For example, if you delete the first two timer objects and create a new object, MATLAB names it `timer-3`, even though the other two timer objects no longer exist in memory. To reset the numeric part of timer object names to 1, execute the `clear classes` command.

Working with Timer Object Properties

In this section...
"Retrieving the Value of Timer Object Properties" on page 10-7
"Setting the Value of Timer Object Properties" on page 10-8

To get information about timer object properties, see the `timer` function reference page.

Retrieving the Value of Timer Object Properties

The timer object supports many properties that provide information about the current state of the timer object and control aspects of its functioning. To retrieve the value of a timer object property, you can use the `get` function or use subscripts (dot notation) to access the field.

The following example uses the `set` function to retrieve the value of the `ExecutionMode` property:

```
t = timer;

tmode = get(t,'ExecutionMode')

tmode =

singleShot
```

The following example uses dot notation to retrieve the value of the `ExecutionMode` property:

```
tmode = t.ExecutionMode

tmode =

singleShot
```

To view a list of all the properties of a timer object, use the `get` function, specifying the timer object as the only argument:

```
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-4'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 0
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
            UserData: []
```

Setting the Value of Timer Object Properties

To set the value of a timer object property, use the `set` function or subscripted assignment (dot notation). You can also set timer object properties when you create the timer object. For more information, see "Creating Timer Objects" on page 10-5.

The following example uses both methods to assign values to timer object properties. The example creates a timer that, once started, displays a message every second until you stop it with the `stop` command.

1 Create a timer object.

```
t = timer;
```

2 Assign values to timer object properties using the `set` function.

```
set(t,'ExecutionMode','fixedRate','BusyMode','drop','Period',1);
```

3 Assign a value to the timer object `TimerFcn` property using dot notation.

```
t.TimerFcn = 'disp(''Processing...'')'
```

4 Start the timer object. It displays a message at 1-second intervals.

```
start(t)
```

5 Stop the timer object.

```
stop(t)
```

6 Delete timer objects after you are done using them.

```
delete(t)
```

Viewing a List of All Settable Properties

To view a list of all timer object properties that can have values assigned to them (in contrast to the read-only properties), use the `set` function, specifying the timer object as the only argument.

The display includes the values you can use to set the property if, like the `BusyMode` property, the property accepts an enumerated list of values.

```
t = timer;

set(t)
   BusyMode: [ {drop} | queue | error ]
   ErrorFcn: string -or- function handle -or- cell array
   ExecutionMode: [{singleShot} | fixedSpacing | fixedDelay | fixedRate]
   Name
   ObjectVisibility: [ {on} | off ]
   Period
   StartDelay
   StartFcn: string -or- function handle -or- cell array
   StopFcn: string -or- function handle -or- cell array
   Tag
   TasksToExecute
   TimerFcn: string -or- function handle -or- cell array
   UserData
```

Starting and Stopping Timers

In this section...

"Starting a Timer" on page 10-10
"Starting a Timer at a Specified Time" on page 10-10
"Stopping Timer Objects" on page 10-11
"Blocking the MATLAB Command Line" on page 10-12

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

Starting a Timer

To start a timer object, call the `start` function, specifying the timer object as the only argument. The `start` function starts a timer object running; the amount of time the timer runs is specified in seconds in the `StartDelay` property.

This example creates a timer object that displays a greeting after 5 seconds elapse.

1 Create a timer object, specifying values for timer object properties.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay', 5);
```

2 Start the timer object.

```
start(t)
```

3 Delete the timer object after you are finished using it.

```
delete(t);
```

Starting a Timer at a Specified Time

To start a timer object and specify a date and time for the timer to fire, (rather than specifying the number of seconds to elapse), use the `startat` function. This function starts a timer object and allows you to specify the date, hour,

minute, and second when you want to the timer to execute. You specify the time as a MATLAB serial date number or as a specially formatted date text string.

This example creates a timer object that displays a message after an hour has elapsed. The `startat` function starts the timer object running and calculates the value of the `StartDelay` property based on the time you specify.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');

startat(t2,now+1/24);
```

Stopping Timer Objects

Once started, the timer object stops running if one of the following conditions apply:

- The timer function callback (`TimerFcn`) has been executed the number of times specified in the `TasksToExecute` property.
- An error occurred while executing a timer function callback (`TimerFcn`).

You can also stop a timer object by using the `stop` function, specifying the timer object as the only argument. The following example illustrates stopping a timer object:

1. Create a timer object.

   ```
   t = timer('TimerFcn','disp(''Hello World!'')', ...
             'StartDelay', 100);
   ```

2. Start it running.

   ```
   start(t)
   ```

3. Check the state of the timer object after starting it.

   ```
   get(t,'Running')

   ans =

   on
   ```

4 Stop the timer using the `stop` command and check the state again. When a timer stops, the value of the `Running` property of the timer object is set to `'off'`.

```
stop(t)

get(t,'Running')

ans =

off
```

5 Delete the timer object when you are finished using it.

```
delete(t)
```

Note The timer object can execute a callback function that you specify when it starts or stops. See "Creating and Executing Callback Functions" on page 10-14.

Blocking the MATLAB Command Line

By default, when you use the `start` or `startat` function to start a timer object, the function returns control to the command line immediately. For some applications, you might prefer to block the command line until the timer fires. To do this, call the `wait` function right after calling the `start` or `startat` function.

1 Create a timer object.

```
t = timer('StartDelay', 5,'TimerFcn', ...
          'disp(''Hello World!'')');
```

2 Start the timer object running.

```
start(t)
```

3 After the `start` function returns, call the `wait` function immediately. The `wait` function blocks the command line until the timer object fires.

```
wait(t)
```

4 Delete the timer object after you are finished using it.

```
delete(t)
```

Creating and Executing Callback Functions

In this section...

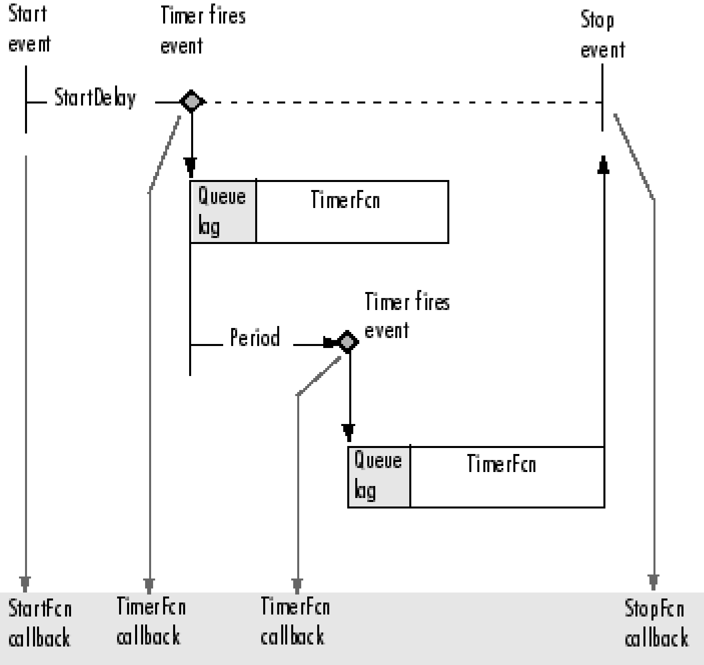
"Associating Commands with Timer Object Events" on page 10-14
"Creating Callback Functions" on page 10-15
"Specifying the Value of Callback Function Properties" on page 10-17

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Associating Commands with Timer Object Events

The timer object supports properties that let you specify the MATLAB commands that execute when a timer fires, and for other timer object events, such as starting, stopping, or when an error occurs. These are called *callbacks*. To associate MATLAB commands with a timer object event, set the value of the associated timer object callback property.

The following diagram shows when the events occur during execution of a timer object and give the names of the timer object properties associated with each event. For example, to associate MATLAB commands with a start event, assign a value to the `StartFcn` callback property. Error callbacks can occur at any time.



Timer Object Events and Related Callback Function

Creating Callback Functions

When the time period specified by a timer object elapses, the timer object executes one or more MATLAB functions of your choosing. You can specify the functions directly as the value of the callback property. You can also put the commands in an M-file function and specify the M-file function as the value of the callback property.

Specifying Callback Functions Directly

This example creates a timer object that displays a greeting after 5 seconds. The example specifies the value of the `TimerFcn` callback property directly, putting the commands in a text string.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay',5);
```

Note When you specify the callback commands directly as the value of the callback function property, the commands are evaluated in the MATLAB workspace.

Putting Commands in a Callback Function

Instead of specifying MATLAB commands directly as the value of a callback property, you can put the commands in an M-file and specify the M-file as the value of the callback property.

When you create a callback function, the first two arguments must be a handle to the timer object and an event structure. An event structure contains two fields: `Type` and `Data`. The `Type` field contains a text string that identifies the type of event that caused the callback. The value of this field can be any of the following strings: `'StartFcn'`, `'StopFcn'`, `'TimerFcn'`, or `'ErrorFcn'`. The `Data` field contains the time the event occurred.

In addition to these two required input arguments, your callback function can accept application-specific arguments. To receive these input arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see "Specifying the Value of Callback Function Properties" on page 10-17.

Example: Writing a Callback Function

This example implements a simple callback function that displays the type of event that triggered the callback and the time the callback occurred. To illustrate passing application-specific arguments, the example callback function accepts as an additional argument a text string and includes this text string in the display output. To see this function used with a callback property, see "Specifying the Value of Callback Function Properties" on page 10-17.

```
function my_callback_fcn(obj, event, string_arg)

txt1 = ' event occurred at ';
txt2 = string_arg;

event_type = event.Type;
event_time = datestr(event.Data.time);
```

```
msg = [event_type txt1 event_time];
disp(msg)
disp(txt2)
```

Specifying the Value of Callback Function Properties

You associate a callback function with a specific event by setting the value of the appropriate callback property. You can specify the callback function as a text string, cell array, or function handle. To access the object and event arguments, you must specify the function as a cell array or as a function handle. If your callback function accepts additional arguments, you must use a cell array.

The following table shows the syntax for several sample callback functions and describes how you call them.

Callback Function Syntax	How to Specify as a Property Value
`function myfile`	`set(h, 'StartFcn', 'myfile')`
`function myfile(obj, event)`	`set(h, 'StartFcn', @myfile)`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {'myfile', 5, 6})`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {@myfile, 5, 6})`

This example illustrates several ways you can specify the value of timer object callback function properties, some with arguments and some without. To see the code of the callback function, `my_callback_fcn`, see "Example: Writing a Callback Function" on page 10-16.

1 Create a timer object.

```
t = timer('StartDelay', 4,'Period', 4,'TasksToExecute', 2,...
          'ExecutionMode','fixedRate');
```

2 Specify the value of the `StartFcn` callback. Note that the example specifies the value in a cell array because the callback function needs to access arguments passed to it.

```
t.StartFcn = {'my_callback_fcn', 'My start message'};
```

3 Specify the value of the `StopFcn` callback. The example specifies the callback function by its handle, rather than as a text string. Again, the value is specified in a cell array because the callback function needs to access the arguments passed to it.

```
t.StopFcn = { @my_callback_fcn, 'My stop message'};
```

4 Specify the value of the `TimerFcn` callback. The example specifies the MATLAB commands in a text string.

```
t.TimerFcn = 'disp(''Hello World!'')';
```

5 Start the timer object.

```
start(t)
```

The example outputs the following.

```
StartFcn event occurred at 10-Mar-2004 17:16:59
Start message
Hello World!
Hello World!
StopFcn event occurred at 10-Mar-2004 17:16:59
Stop message
```

6 Delete the timer object after you are finished with it.

```
delete(t)
```

Timer Object Execution Modes

In this section...

"Executing a Timer Callback Function Once" on page 10-19

"Executing a Timer Callback Function Multiple Times" on page 10-20

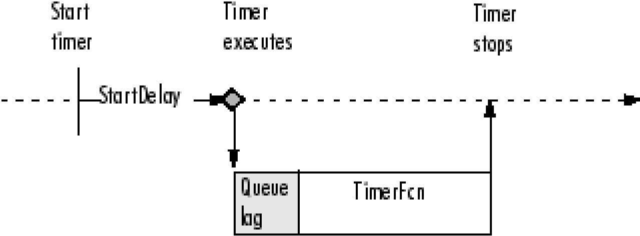
"Handling Callback Function Queuing Conflicts" on page 10-21

Executing a Timer Callback Function Once

The timer object supports several execution modes that determine how it schedules the timer callback function (`TimerFcn`) for execution. You specify the execution mode by setting the value of the `ExecutionMode` property.

To execute a timer callback function once, set the `ExecutionMode` property to `'singleShot'`. This is the default execution mode. In this mode, the timer object starts the timer and, after the time period specified in the `StartDelay` property elapses, adds the timer callback function (`TimerFcn`) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

The following figure graphically illustrates the parts of timer callback execution for a `singleShot` execution mode. The shaded area in the figure, labelled queue lag, represents the indeterminate amount of time between when the timer adds a timer callback function to the MATLAB execution queue and when the function starts executing. The duration of this lag is dependent on what other processing MATLAB happens to be doing at the time.



Timer Callback Execution (singleShot Execution Mode)

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- 'fixedRate'
- 'fixedDelay'
- 'fixedSpacing'

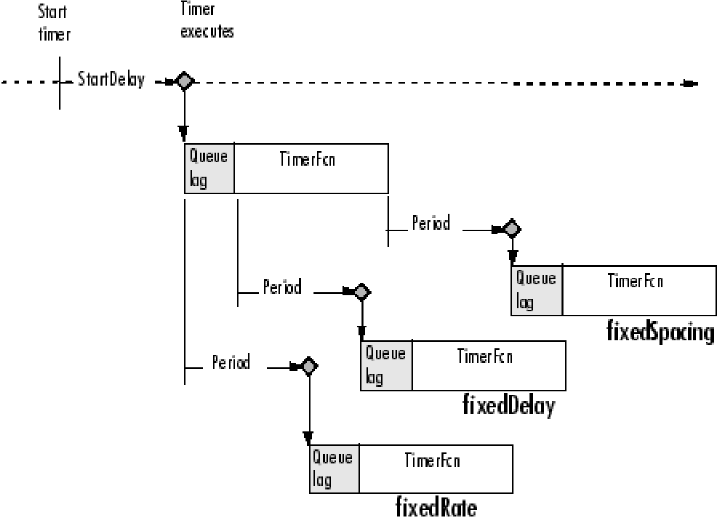
In many ways, these execution modes operate the same:

- The TasksToExecute property specifies the number of times you want the timer to execute the timer callback function (TimerFcn).
- The Period property specifies the amount of time between executions of the timer callback function.
- The BusyMode property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See “Handling Callback Function Queuing Conflicts” on page 10-21 for more information.

The execution modes differ only in where they start measuring the time period between executions. The following table describes these differences.

Execution Mode	Description
'fixedRate'	Time period between executions begins immediately after the timer callback function is added to the MATLAB execution queue.
'fixedDelay'	Time period between executions begins when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue.
'fixedSpacing'	Time period between executions begins when the timer callback function finishes executing.

The following figure illustrates the difference between these modes. Note that the amount of time between executions (specified by the Period property) remains the same. Only the point at which execution begins is different.



Differences Between Execution Modes

Handling Callback Function Queuing Conflicts

At busy times, in multiple-execution scenarios, the timer may need to add the timer callback function (`TimerFcn`) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the `BusyMode` property.

If you specify `'drop'` as the value of the `BusyMode` property, the timer object skips the execution of the timer function callback if the previously scheduled callback function has not already completed.

If you specify `'queue'`, the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

Note In 'queue' mode, the timer object tries to make the average time between executions equal the amount of time specified in the Period property. If the timer object has to wait longer than the time specified in the Period property between executions of the timer function callback, it shortens the time period for subsequent executions to make up the time.

If the BusyMode property is set to 'error', the timer object stops and executes the timer object error callback function (ErrorFcn), if one is specified.

Deleting Timer Objects from Memory

In this section...
“Deleting One or More Timer Objects” on page 10-23
“Testing the Validity of a Timer Object” on page 10-23

Deleting One or More Timer Objects

When you are finished with a timer object, delete it from memory using the `delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object remain. Deleted timer objects are invalid and cannot be reused. Use the `clear` command to remove workspace variables that reference deleted timer objects.

To remove all timer objects from memory, enter

```
delete(timerfind)
```

For information about the `timerfind` function, see “Finding Timer Objects in Memory” on page 10-24.

Testing the Validity of a Timer Object

To test if a timer object has been deleted, use the `isvalid` function. The `isvalid` function returns logical `0` (`false`) for deleted timer objects:

```
isvalid(t)
ans =

      0
```

Finding Timer Objects in Memory

In this section...
"Finding All Timer Objects" on page 10-24
"Finding Invisible Timer Objects" on page 10-24

Finding All Timer Objects

To find all the timer objects that exist in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t1 = timer;
t2 = timer;
t3 = timer;
t_array = timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:    Name:
   1       singleShot      1        ''           timer-3
   2       singleShot      1        ''           timer-4
   3       singleShot      1        ''           timer-5
```

Using `timerfind` to determine all the timer objects that exist in memory can be helpful when deleting timer objects.

Finding Invisible Timer Objects

If you set the value of a timer object's `ObjectVisibility` property to `'off'`, the timer object does not appear in listings of existing timer objects returned by `timerfind`. The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the timer objects created by their application.

Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties. To

retrieve a list of all the timer objects in memory, including invisible ones, use the `timerfindall` function.

Improving Performance and Memory Usage

Analyzing Your Program's Performance (p. 11-2)	What tools are provided by MATLAB to measure the performance of your programs and identify where the bottlenecks are
Techniques for Improving Performance (p. 11-4)	How to improve M-file performance by vectorizing loops, preallocating arrays, etc.
Multiprocessing in MATLAB (p. 11-13)	How to speed up performance with two types of multiprocessing provided with MATLAB
Memory Allocation in MATLAB (p. 11-18)	How MATLAB allocates memory to different data structures, array types, etc.
Memory Management Functions (p. 11-24)	MATLAB functions that can help you manage memory use
Strategies for Efficient Use of Memory (p. 11-25)	Tips to help you reduce fragmentation and allocate memory more efficiently
Resolving "Out of Memory" Errors (p. 11-27)	What to do when if you get an "Out of Memory" error

Analyzing Your Program's Performance

In this section...

"Overview" on page 11-2

"The M-File Profiler Utility" on page 11-2

"Stopwatch Timer Functions" on page 11-2

Overview

The M-file Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement. The Profiler can be more useful in measuring relative execution time and in identifying specific performance bottlenecks in your code, while the stopwatch functions tend to be more useful for providing absolute time measurements.

The M-File Profiler Utility

A good first step to speeding up your programs is to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

MATLAB provides the M-file Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type `profile viewer` or select **Desktop > Profiler** in the MATLAB Command Window. See Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation, and the `profile` function reference page.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, `tic` and `toc`. Invoking

tic starts the timer, and the first subsequent toc stops it and reports the time elapsed between the two.

Use tic and toc as shown here:

```
tic
   -- run the program section to be timed --
toc
```

Keep in mind that tic and toc measure overall elapsed time. Make sure that no other applications are running in the background on your system that could affect the timing of your MATLAB programs.

Measuring Smaller Programs

Shorter programs sometimes run too fast to get useful data from tic and toc. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run:

```
tic
   for k = 1:100
      -- run the program --
   end
toc
```

Using tic and toc Versus the cputime Function

Although it is possible to measure performance using the cputime function, it is recommended that you use the tic and toc functions for this purpose exclusively. It has been the general rule for CPU-intensive calculations run on Windows machines that the elapsed time using cputime and the elapsed time using tic and toc are close in value, ignoring any first time costs. There are cases however that show a significant difference between these two methods. For example, in the case of a Pentium 4 with hyperthreading running Windows, there can be a significant difference between the values returned by cputime versus tic and toc.

Techniques for Improving Performance

In this section...

"Vectorizing Loops" on page 11-4

"Preallocating Arrays" on page 11-7

"Use Distributed Arrays for Large Datasets" on page 11-9

"When Possible, Replace for with parfor (Parallel for)" on page 11-9

"Multithreading Capabilities in MATLAB" on page 11-9

"Limiting M-File Size and Complexity" on page 11-9

"Coding Loops in a MEX-File" on page 11-10

"Assigning to Variables" on page 11-10

"Operating on Real Data" on page 11-11

"Using Appropriate Logical Operators" on page 11-11

"Overloading Built-In Functions" on page 11-12

"Functions Are Generally Faster Than Scripts" on page 11-12

"Load and Save Are Faster Than File I/O Functions" on page 11-12

"Avoid Large Background Processes" on page 11-12

Vectorizing Loops

MATLAB is a matrix language, which means it is designed for vector and matrix operations. You can often speed up your M-file code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Simple Example of Vectorizing

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
```

```
end
```

A vectorized version of the same code is

```
t = 0:.01:10;
y = sin(t);
```

The second example executes much faster than the first and is the way MATLAB is meant to be used. Test this on your system by creating M-file scripts that contain the code shown, and then using the `tic` and `toc` functions to time the M-files.

Advanced Example of Vectorizing

`repmat` is an example of a function that takes advantage of vectorization. It accepts three input arguments: an array `A`, a row dimension `M`, and a column dimension `N`.

`repmat` creates an output array that contains the elements of array `A`, replicated and "tiled" in an `M`-by-`N` arrangement:

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

B = repmat(A,2,3);
B =
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
```

`repmat` uses vectorization to create the indices that place elements in the output array:

```
function B = repmat(A, M, N)

% Step 1 Get row and column sizes
[m,n] = size(A);

% Step 2 Generate vectors of indices from 1 to row/column size
mind = (1:m)';
nind = (1:n)';
```

```
% Step 3 Create index matrices from vectors above
mind = mind(:,ones(1, M));
nind = nind(:,ones(1, N));

% Step 4 Create output array
B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. `mind` contains the integers from 1 through the row size of `A`. The `nind` variable contains the integers from 1 through the column size of `A`.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,nCols))
```

where `nCols` is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, `mind`, is paired with each element of the column index array, `nind`, using the following procedure:

1. The first element of `mind`, the row index, is paired with each element of `nind`. MATLAB moves through the `nind` matrix in a columnwise fashion, so `mind(1,1)` goes with `nind(1,1)`, and then `nind(2,1)`, and so on. The result fills the first row of the output array.

2. Moving columnwise through `mind`, each element is paired with the elements of `nind` as above. Each complete pass through the `nind` matrix fills one row of the output array.

Caution While `repmat` can take advantage of vectorization, it can do so at the expense of memory usage. When this is the case, you might find the `bsxfun` function be more appropriate in this respect.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are as follows

Function	Description
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`cumsum`	Find cumulative sum
`diff`	Find differences and approximate derivatives
`find`	Find indices and values of nonzero elements
`ind2sub`	Convert from linear index to subscripts
`ipermute`	Inverse permute dimensions of a multidimensional array
`logical`	Convert numeric values to logical
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`permute`	Rearrange dimensions of a multidimensional array
`prod`	Find product of array elements
`repmat`	Replicate and tile an array
`reshape`	Change the shape of an array
`shiftdim`	Shift array dimensions
`sort`	Sort array elements in ascending or descending order
`squeeze`	Remove singleton dimensions from an array
`sub2ind`	Convert from subscripts to linear index
`sum`	Find the sum of array elements

Preallocating Arrays

`for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve on code execution time by

preallocating the maximum amount of space that would be required for the array ahead of time.

The following code creates a scalar variable `x`, and then gradually increases the size of `x` in a `for` loop instead of preallocating the required amount of memory at the start:

```
x = 0;
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Change the first line to preallocate a 1-by-1000 block of memory for `x` initialized to zero. This time there is no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop:

```
x = zeros(1, 1000);
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Preallocation Functions

Preallocation makes it unnecessary for MATLAB to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type	Function	Examples
Numeric	`zeros`	`y = zeros(1, 100);`

Cell	`cell`	`B = cell(2, 3);`
		`B{1,3} = 1:3;`
		`B{2,2} = 'string';`

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, avoid using the method

```
A = int8(zeros(100));
```

This statement preallocates a 100-by-100 matrix of `int8` first by creating a full matrix of `doubles`, and then converting each element to `int8`. This costs time and uses memory unnecessarily.

The next statement shows how to do this more efficiently:

```
A = zeros(100, 'int8');
```

Use Distributed Arrays for Large Datasets

This topic is described in the "Parallel Math" section of the Distributed Computing Toolbox documentation.

When Possible, Replace for with parfor (Parallel for)

This topic is described in the "Parallel for-Loops" section of the Distributed Computing Toolbox documentation.

Multithreading Capabilities in MATLAB

See "Implicit Multiprocessing" on page 11-14 to learn more about making use of multithreaded computation.

Limiting M-File Size and Complexity

Running programs that are unusually large or complex can put a strain on your system's resources. For example, a program that nearly exceeds memory capacity may work some of the time and sometimes not, depending on the commands it uses and on what other applications are running at the time. An example of unnecessary complexity might be having a large number of `if` and `else` statements where `switch` and `case` might be more suitable. This can also lead to performance and space problems.

If you have an M-file that includes thousands of variables or functions, tens of thousands of statements, or hundreds of language keyword pairs (e.g., `if-else`, or `try-catch`), then making some of the changes suggested here is likely to not only boost its performance and reliability, but should make your program code easier to understand and maintain as well.

Here are a few suggestions on how to make your programs less resource-intensive:

- Split large script files into smaller ones, having the first file call the second if necessary.
- Take your larger chunks of program code and make separate functions (or subfunctions and nested functions) of them.
- If you have functions or expressions by that seem overly complicated, make smaller and simpler functions or expressions of them. Simpler functions are also more likely to be made into utility functions that you can share with others.

Coding Loops in a MEX-File

If there are instances where you cannot vectorize and must use a `for` or `while` loop, consider coding the loop in a MEX-file. In this way, the loop executes much more quickly since the instructions in the loop do not have to be interpreted each time they execute.

See “Introducing MEX-Files” in the External Interfaces documentation.

Assigning to Variables

For best performance, keep the following suggestions in mind when assigning values to variables.

Changing a Variable’s Data Type or Dimension

Changing the data type or array shape of an existing variable slows MATLAB down as it must take extra time to process this. When you need to store data of a different type, it is advisable to create a new variable.

This code changes the type for `X` from `double` to `char`, which has a negative impact on performance:

```
X = 23;
     .
-- other code  --
     .
X = 'A';                    % X changed from type double to char
     .
-- other code  --
```

Assigning Real and Complex Numbers

Assigning a complex number to a variable that already holds a real number impacts the performance of your program. Similarly, you should not assign a real value to a variable that already holds a complex value.

Operating on Real Data

When operating on real (i.e., noncomplex) numbers, it is more efficient to use MATLAB functions that have been designed specifically for real numbers. The following functions return numeric values that are real.

Function	Description
`reallog`	Find natural logarithm for nonnegative real arrays
`realpow`	Find array power for real-only output
`realsqrt`	Find square root for nonnegative real arrays

Using Appropriate Logical Operators

When performing a logical AND or OR operation, you have a choice of two operators of each type.

Operator	Description
`&`, `|`	Perform logical AND and OR on arrays element by element
`&&`, `||`	Perform logical AND and OR on scalar values with short-circuiting

In `if` and `while` statements, it is more efficient to use the short-circuiting operators, `&&` for logical AND and `||` for logical OR. This is because these operators often don't have to evaluate the entire logical expression. For example, MATLAB evaluates only the first part of this expression whenever the number of input arguments is less than three:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

See Short-Circuit Operators in the MATLAB documentation for a discussion on short-circuiting with `&&` and `||`.

Overloading Built-In Functions

Overloading MATLAB built-in functions on any of the standard MATLAB data types can negatively affect performance. For example, if you overload the `plus` function to handle any of the integer data types differently, you may hinder certain optimizations in the MATLAB built-in function code for `plus`, and thus may slow down any programs that make use of this overload.

Functions Are Generally Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the low-level MATLAB file I/O routines such as `fread` and `fwrite`, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

Multiprocessing in MATLAB

In this section...

"Overview" on page 11-13
"Implicit Multiprocessing" on page 11-14
"Explicit Multiprocessing" on page 11-17

Overview

MATLAB supports two types of multiprocessing: *implicit* and *explicit*.

Implicit Multiprocessing

Characteristics of implicit multiprocessing:

- Runs multiple threads on a single machine, most often using one thread per processing unit.
- Requires a multiple CPU (multiprocessor or multicore) system.
- Speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.
- Does *not* require any changes to your MATLAB code.
- Works behind the scenes to take advantage of the processing units available to you. It does this by multithreading the computationally-intensive math library functions that you use in the course of your MATLAB session.

Enable implicit multiprocessing with the MATLAB Preferences Panel to enable or disable, or to set the number of threads to be used. You can change the maximum number of threads programmatically using the `maxNumCompThreads` function.

Explicit Multiprocessing

Characteristics of explicit multiprocessing:

- Runs separate processes on one or many machines.

- Requires installation of Distributed Computing Toolbox (DCT).
- Speeds up execution of large MATLAB jobs. Enables you to run jobs simultaneously on a cluster of computers, or as several processes on a single machine.
- Requires that you modify your MATLAB code.
- DCT supports programming constructs for distributed arrays and parallel `for` (`parfor`) loops. It also supports both interactive and batch execution.

Enable explicit multiprocessing by installing Distributed Computing Toolbox.

Implicit Multiprocessing

Multithreaded computation runs in a single instance of MATLAB and generates simultaneous instruction streams on a multiple CPU (multiprocessor or multicore) system. The multiple processors share the memory of a single computer. The work to be processed is implicitly partitioned for execution on multiple threads. In particular, multithreaded computation in MATLAB speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.

If you are using a multiple-CPU system, you can run a demo to see the performance impact—see Multithreaded Computation in the Help browser **Demos** pane, under MATLAB Mathematics.

For information regarding specific functions, search for “What MATLAB Functions Support Multithreaded Computation” on The MathWorks online Support page.

Platform Differences and Multithreaded Computation

The BLAS library used for multithreaded computation differs according to which platform you are using:

Platform	BLAS Used
Windows with Intel processors	Intel MKL BLAS
Windows with AMD processors	AMD ACML BLAS
Linux with Intel processors	Intel MKL BLAS
Linux with AMD processors	AMD ACML BLAS
Macintosh Intel-based	Intel MKL BLAS
MacIntosh PowerPC	Mac Accelerate BLAS
Solaris	Sun Performance Library BLAS

Note On Macintosh PowerPC platforms, multithreaded computation is always enabled for the Accelerate BLAS. To enable multithreaded computation for elementwise operations, use MATLAB preferences.

Enabling Multithreaded Computation

The preference automatically detects the number of CPUs on your system and recommends the number of threads based on that.

Multithreaded computation in MATLAB is disabled by default. To enable it and set the maximum number of threads to use, follow these steps:

1. Select **File > Preferences > General > Multithreading**.

   The **General Multithreading Preferences** panel opens.

2. On the **General Multithreading Preferences** panel, select **Enable multithreaded computation**.

3. Specify the **Maximum number of computational threads**. Accepting the **Automatic** option is recommended—MATLAB automatically sets the value to the actual number of computational cores on your system. Note

that if your system uses hyperthreading (where one processor is logically configured as two), MATLAB sets the value to 1.

If you choose **Manual**, enter the maximum number of threads you want to set; use a positive integer not greater than 16. (Selecting a number other than the recommended value might increase performance for some computations, but might decrease performance for others.)

Note You may find that, at certain times, a library function uses a number of threads smaller than what you have specified. This can happen if the function finds the specified number of threads to be inappropriate.

In the event of an abnormal termination with multithreaded computation enabled, MATLAB behaves differently than when multithreaded computation is not enabled. For details, see .

Making this setting in the **Preferences** panel not only affects your current MATLAB session, but future sessions as well. To disable multithreaded computation, clear the **Enable multithreaded computation** selection and click **OK**.

Note For Macintosh PowerPC platforms, BLAS multithreaded computation cannot be disabled.

Setting the Number of Threads Programmatically

To set or retrieve the maximum number of computational threads from within an M-file program, use the `maxNumCompThreads` function. You can either set the maximum number of computational threads to a specific number, or indicate that you want the setting to be done automatically by MATLAB.

To set the maximum number of computational threads to a specific number `N`, use

```
maxNumCompThreads(N)
```

To have MATLAB set the maximum number of threads, use:

```
maxNumCompThreads('automatic')
```

`maxNumCompThreads` also returns the current maximum number of threads if you call it with an output value:

```
old_N = MaxNumCompThreads(new_N)
```

MATLAB keeps the settings you make using `maxNumCompThreads` synchronous with your **Preferences** settings. If you change the maximum number of computational threads by means of the `maxNumCompThreads` function, MATLAB updates the **Preferences** panel to agree with the new setting.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session. To make this setting carry over to future sessions, use the **Preferences** panel instead.

Crash Recovery and Multithreading

If MATLAB experiences a segmentation violation or other serious problem when multithreaded computation is enabled, it cannot try to return control to the Command Window. You do not have an opportunity to view a segmentation violation message in the Command Window as you might when multithreaded computation is *not* enabled. Instead, your platform's vendor, for example, Microsoft or Apple, provides an error dialog box. MATLAB then terminates.

Upon the next MATLAB startup after a fatal problem, the "Error Log Reporter" prompts you to e-mail the log to The MathWorks.

Explicit Multiprocessing

See the Distributed Computing Toolbox documentation for information regarding explicit multiprocessing in MATLAB.

Memory Allocation in MATLAB

In this section...

"Memory Allocation for Arrays" on page 11-18

"Data Structures and Memory" on page 11-22

For more information on memory management, see Technical Note 1106: "Memory Management Guide" at the following URL:

```
http://www.mathworks.com/support/tech-notes/1100/1106.html
```

Memory Allocation for Arrays

The topics below provide information on how MATLAB allocates memory when working with arrays and variables. The purpose is to help you use memory more efficiently when writing code. Most of the time, however, you should not need to be concerned with these internal operations as MATLAB handles data storage for you automatically.

- "Creating and Modifying Arrays" on page 11-18
- "Copying Arrays" on page 11-19
- "Array Headers" on page 11-20
- "Function Arguments" on page 11-21

Note Any information on how data is handled internally by MATLAB is subject to change in future releases.

Creating and Modifying Arrays

When you assign any type of data (a numeric, string, or structure array, for example) to a variable, MATLAB allocates a contiguous block of memory and stores the array data in that block. It also stores information about the array data, such as its data type and dimensions, in a separate, small block of memory called a header. The variable that you assign this data to is actually a *pointer* to the data; it does not *contain* the data.

If you add new elements to an existing array, MATLAB expands the existing array in memory in a way that keeps its storage contiguous. This might require finding a new block of memory large enough to hold the expanded array, and then copying the contents of the array from its original location to the new block in memory, adding the new elements to the array in this block, and freeing up the original array location in memory.

If you remove elements from an existing array, MATLAB keeps the memory storage contiguous by removing the deleted elements, and then compacting its storage in the original memory location.

Working with Large Data Sets. If you are working with large data sets, you need to be careful when increasing the size of an array to avoid getting errors caused by insufficient memory. If you expand the array beyond the available contiguous memory of its original location, MATLAB has to make a copy of the array in a new location in memory, as explained above, and then set this array to its new value. During this operation, there are two copies of the original array in memory, thus temporarily doubling the amount of memory required for the array and increasing the risk of your program running out of memory during execution. It is better to preallocate sufficient memory for the array at the start. See "Preallocating Arrays" on page 11-7.

Copying Arrays

Internally, multiple variables can point to the same block of data, thus sharing that array's value. When you copy a variable to another variable (e.g., `B = A`), MATLAB makes a copy of the pointer, not the array. For example, the following code creates a single 500-by-500 matrix and two pointers to it, `A` and `B`:

```
A = magic(500);
B = A;
```

As long as the contents of the array are not modified, there is no need to store two copies of it. If you modify the array, then MATLAB does create a separate array to hold the new values.

If you modify the array shown above by referencing it with variable `A` (e.g., `A(400,:) = 0`), then MATLAB creates a copy of the array, modifies it accordingly, and stores a pointer to the new array in `A`. Variable `B` continues

to point to the original array. If you modify the array by referencing it with variable `B` (e.g., `B(400,:) = 0`), the same thing happens except that it is `B` that points to the new array.

Array Headers

When you assign an array to a variable, MATLAB also stores information about the array (such as data type and dimensions) in a separate piece of memory called a header. For most arrays, the memory required to store the header is insignificant. There is a small advantage though to storing large data sets in a small number of large arrays as opposed to a large number of small arrays, as the former configuration requires fewer array headers.

Structure and Cell Arrays. For structures and cell arrays, MATLAB creates a header not only for each array, but also for each field of the structure and for each cell of a cell array. Because of this, the amount of memory required to store a structure or cell array depends not only on how much data it holds, but also how it is constructed.

For example, a scalar structure array `S1` having fields `R`, `G`, and `B`, each field of size 100-by-50, requires one array header to describe the overall structure, and one header to describe each of the three field arrays, making a total of 4 array headers for the entire data structure:

```
S1.R(1:100,1:50)
S1.G(1:100,1:50)
S1.B(1:100,1:50)
```

On the other hand, a 100-by-50 structure array `S2` in which each element has scalar fields `R`, `G`, and `B` requires one array header to describe the overall structure, and one array header per field for each of the 5,000 elements of the structure, making a total of 15,001 array headers for the entire data structure:

```
S2(1:100,1:50).R
S2(1:100,1:50).G
S2(1:100,1:50).B
```

Thus, even though `S1` and `S2` contain the same amount of data, S1 uses significantly less space in memory. Not only is less memory required, but there is a corresponding speed benefit to using the `S1` format as well.

Memory Usage Reported By the whos Function. The whos function displays the amount of memory consumed by any variable. For reasons of simplicity, whos reports only the memory used to store the actual data. It does not report storage for the variable itself or the array header.

Function Arguments

MATLAB handles arguments passed in function calls in a similar way. When you pass a variable to a function, you are actually passing a pointer to the data that the variable represents. As long as the input data is not modified by the function being called, the variable in the calling function and the variable in the called function point to the same location in memory. If the called function modifies the value of the input data, then MATLAB makes a copy of the original array in a new location in memory, updates that copy with the modified value, and points the input variable in the called function to this new array.

In the example below, function myfun modifies the value of the array passed into it. MATLAB makes a copy in memory of the array pointed to by A, sets variable X as a pointer to this new array, and then sets one row of X to zero. The array referenced by A remains unchanged:

```
A = magic(500);
myfun(A);

function myfun(X)
X(400,:) = 0;
```

If the calling function needs the modified value of the array it passed to myfun, you will need to return the updated array as an output of the called function, as shown here for variable A:

```
A = magic(500);
A = myfun(A);
sprintf('The new value of A is %d', A)

function Y = myfun(X)
X(400,:) = 0;
Y = X;
```

Working with Large Data Sets. Again, when working with large data sets, you should be aware that MATLAB makes a temporary copy of A if the called function modifies its value. This temporarily doubles the memory required to store the array, which causes MATLAB to generate an error if sufficient memory is not available.

One way to avoid running out of memory in this situation is to use nested functions. A nested function shares the workspace of all outer functions, giving the nested function access to data outside of its usual scope. In the example shown here, nested function `setrowval` has direct access to the workspace of the outer function `myfun`, making it unnecessary to pass a copy of the variable in the function call. When `setrowval` modifies the value of A, it modifies it in the workspace of the calling function. There is no need to use additional memory to hold a separate array for the function being called, and there also is no need to return the modified value of A:

```
function myfun
A = magic(500);

   function setrowval(row, value)
   A(row,:) = value;
   end

setrowval(400, 0);
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)
end
```

Data Structures and Memory

Memory requirements differ for the various types of MATLAB data structures. You may be able to reduce the amount of memory used for these structures by considering how MATLAB stores them.

Numeric Arrays

MATLAB requires 1, 2, 4, or 8 bytes to store 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers respectively. For floating-point numbers, MATLAB uses 4 or 8 bytes for `single` and `double` types. To conserve memory, The MathWorks recommends that you use the smallest integer or floating-point type that will contain your data without overflowing. For more

information, see "Numeric Types" in the MATLAB Programming section on Data Types.

Complex Arrays

MATLAB stores complex data as separate real and imaginary parts. If you make a copy of a complex array variable, and then modify only the real or imaginary part of the array, MATLAB creates a new array containing both real and imaginary parts.

Sparse Matrices

It is best to store matrices with values that are mostly zero in sparse format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse format using the `sparse` function.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. As shown below, approximately half as much memory is required for the sparse matrix:

```
whos
  Name      Size                    Bytes  Class

  X      1000x1000                8000000  double array
  Y      1000x1000                4004000  double array (sparse)
```

Memory Management Functions

The following functions can help you to manage memory use in MATLAB:

- `whos` shows how much memory has been allocated for variables in the workspace.
- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.

  See "Compressing Data in Memory" on page 11-28.
- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically clear variables from memory that you no longer need.

  If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you are no longer using from memory. Use `clear` to do this.
- `save` selectively stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then use the `clear` function to remove the saved data from memory.
- `load` reloads a data file saved with the `save` function.
- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on UNIX systems as UNIX does not free up memory allocated to an application (e.g., MATLAB) until the application exits.

You can use the `save` and `load` functions in conjunction with the `quit` command to free memory by:

1. Saving any needed variables with the `save` function.
2. Quitting MATLAB to free all memory allocated to MATLAB.
3. Starting a new MATLAB session and loading the saved variables back into the clean MATLAB workspace.

Strategies for Efficient Use of Memory

In this section...

"Preallocating Arrays to Reduce Fragmentation" on page 11-25

"Allocating Large Matrices Earlier" on page 11-26

"Working with Large Amounts of Data" on page 11-26

To conserve memory when creating variables,

- Avoid creating large temporary variables, and clear temporary variables when they are no longer needed.
- When working with arrays of fixed size, preallocate them rather than having MATLAB resize the array each time you enlarge it.
- Allocate your larger matrices first, as explained in .
- Set variables equal to the empty matrix `[]` to free memory, or clear the variables using the `clear` function.
- Reuse variables as much as possible, but keeping in mind the guidelines stated in "Assigning to Variables" on page 11-10.

Preallocating Arrays to Reduce Fragmentation

In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. `for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can add to this fragmentation as they have to repeatedly find and allocate larger blocks of memory to store the data.

To make more efficient use of your memory, preallocate a block of memory large enough to hold the matrix at its final size before entering the loop. When you preallocate memory for a potentially large array, MATLAB "grabs" sufficient contiguous space for the data at the beginning of the computation. Once you have this space, you can add elements to the array without having to continually allocate new space for it in memory.

For more information on preallocation, see "Preallocating Arrays" on page 11-7.

Allocating Large Matrices Earlier

MATLAB uses a heap method of memory management. It requests memory from the operating system when there is not enough memory available in the MATLAB heap to store the current variables. It reuses memory as long as the size of the memory segment required is available in the MATLAB heap.

For example, on one machine these statements use approximately 15.4 MB of RAM:

```
a = rand(1e6,1);
b = rand(1e6,1);
```

This statement uses approximately 16.4 MB of RAM:

```
c = rand(2.1e6,1);
```

The following statements can use approximately 32.4 MB of RAM. This is because MATLAB may not be able to reuse the space previously occupied by two 1MB arrays when allocating space for a 2.1 MB array:

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.1e6,1);
```

The simplest way to prevent overallocation of memory is to allocate the largest vectors first. These statements use only about 16.4 MB of RAM:

```
c = rand(2.1e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

Working with Large Amounts of Data

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, use the `clear` function to remove the variable from memory and continue with the data generation.

Resolving "Out of Memory" Errors

In this section...

"General Suggestions for Reclaiming Memory" on page 11-27
"Compressing Data in Memory" on page 11-28
"Increasing System Swap Space" on page 11-28
"Freeing Up System Resources on Windows Systems" on page 11-29
"Reloading Variables on UNIX Systems" on page 11-30

For more information on this topic, search for "Avoiding Out of Memory Errors" on The MathWorks online "Support" page.

General Suggestions for Reclaiming Memory

MATLAB generates an `Out of Memory` message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see the `Out of Memory` message, use any of the techniques discussed under "Memory Allocation in MATLAB" on page 11-18 to help optimize the available memory. If the `Out of Memory` message still appears, you can try any of the following:

- Compress data to reduce memory fragmentation
- If possible, break large matrices into several smaller matrices so that less memory is used at any one time.
- If possible, reduce the size of your data.
- Make sure that there are no external constraints on the memory accessible to MATLAB. (On UNIX systems, use the `limit` command to check).
- Increase the size of the swap file. We recommend that your machine be configured with twice as much swap space as you have RAM. See "Increasing System Swap Space" on page 11-28, below.
- Add more memory to the system.

On machines running Windows 2000 Advanced Server or Windows 2000 Datacenter Server, the amount of virtual memory space reserved by the

operating system can be reduced by using the /3GB switch in the `boot.ini` file. More documentation on this option can be found at the following URL:

http://support.microsoft.com/support/kb/articles/Q291/9/88.ASP

Similarly, on machines running Windows Vista, you can achieve the same effect by using the command:

```
BCDEdit /set increaseuserva 3072
```

More documentation on this option can be found at the following URL:

http://msdn2.microsoft.com/en-us/library/aa906211.aspx

Compressing Data in Memory

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. If you get the `Out of Memory` message from MATLAB, the `pack` function may be able to compress some of your data in memory, thus freeing up larger contiguous blocks.

Note Because of time considerations, you should not use `pack` within loops or M-file functions.

Increasing System Swap Space

How you set the swap space for your computer depends on what operating system you are running on.

UNIX

Information about swap space can be procured by typing `pstat -s` at the UNIX command prompt. For detailed information on changing swap space, ask your system administrator.

Linux

Swap space can be changed by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man` followed by the command name at the Linux prompt.

Windows 2000

Follow the steps shown here:

1. Right-click the **My Computer** icon, and select **Properties**.
2. Select the **Advanced** tab and click the **Performance Options** button.
3. Click the **Change** button to change the amount of virtual memory.

Windows XP

Follow the steps shown here:

1. Right-click the **My Computer** icon, and select **Properties**.
2. In the **System Properties** GUI, select the **Advanced** tab. In the section labeled **Performance**, click the **Settings** button.
3. In the **Performance Options** GUI, click **Advanced**. In the section labeled **Virtual Memory**, click the **Change** button
4. In the **Virtual Memory** GUI, under **Paging file size for selected drive**, you can change the amount of virtual memory.

Freeing Up System Resources on Windows Systems

There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows uses system resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UI controls. One way to free up system resources is to close all inactive windows. Windows icons still use resources.

Reloading Variables on UNIX Systems

On UNIX systems, MATLAB does not return memory to the operating system even after variables have been cleared. This is due to the manner in which UNIX manages memory. UNIX does not accept memory back from a program until the program has terminated. So, the amount of memory used in a MATLAB session is not returned to the operating system until you exit MATLAB.

To free up the memory used in your MATLAB session, save your workspace variables, exit MATLAB, and then load your variables back in.

12

Programming Tips

Introduction (p. 12-3)	How to Use the Programming Tips
Command and Function Syntax (p. 12-4)	Syntax, command shortcuts, command recall, etc.
Help (p. 12-7)	Getting help on MATLAB functions and your own
Development Environment (p. 12-12)	Useful features in the development environment
M-File Functions (p. 12-14)	M-file structure, getting information about a function
Function Arguments (p. 12-17)	Various ways to pass arguments, useful functions
Program Development (p. 12-20)	Suggestions for creating and modifying program code
Debugging (p. 12-23)	Using the debugging environment and commands
Variables (p. 12-27)	Variable names, global and persistent variables
Strings (p. 12-31)	String concatenation, string conversion, etc.
Evaluating Expressions (p. 12-34)	Use of eval, short-circuiting logical expressions, etc.
MATLAB Path (p. 12-36)	Precedence rules, making file changes visible to MATLAB, etc.
Program Control (p. 12-40)	Using program control statements like if, switch, try

Save and Load (p. 12-44)	Saving MATLAB data to a file, loading it back in
Files and Filenames (p. 12-47)	Naming M-files, passing filenames, etc.
Input/Output (p. 12-50)	Reading and writing various types of files
Starting MATLAB (p. 12-53)	Getting MATLAB to start up faster
Operating System Compatibility (p. 12-54)	Interacting with the operating system
Demos (p. 12-56)	Learning about the demos supplied with MATLAB
For More Information (p. 12-57)	Other valuable resources for information

Introduction

This section is a categorized compilation of tips for the MATLAB® programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see Chapter 11, “Improving Performance and Memory Usage”

Command and Function Syntax

In this section...

"Syntax Help" on page 12-4

"Command and Function Syntaxes" on page 12-4

"Command Line Continuation" on page 12-4

"Completing Commands Using the Tab Key" on page 12-5

"Recalling Commands" on page 12-5

"Clearing Commands" on page 12-6

"Suppressing Output to the Screen" on page 12-6

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3                 % Command syntax
functionname('arg1','arg2','arg3')          % Function syntax
```

For more information: See "Calling Functions" on page 4-52in the MATLAB Programming documentation.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (...). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
         exampleNumber, ...
```

```
            numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...
      to another line, resulting in an error.'
```

For more information: See Entering Long Lines in the MATLAB Desktop Tools and Development Environment documentation.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;
set(f, 'papTuT,'cT)                   % Type this line.
set(f, 'paperunits','centimeters')    % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT
PaperOrientation   PaperPositionMode  PaperType     Parent
PaperPosition      PaperSize          PaperUnits
```

For more information: See Tab Completion in the MATLAB Desktop Tools and Development Environment documentation

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.
- Open the Command History window (**View > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Recalling Previous Lines and Command History in the MATLAB Desktop Tools and Development Environment documentation.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);     % Create matrix A, but do not display it.
```

Help

In this section...

"Using the Help Browser" on page 12-7

"Help on Functions from the Help Browser" on page 12-8

"Help on Functions from the Command Window" on page 12-8

"Topical Help" on page 12-8

"Paged Output" on page 12-9

"Writing Your Own Help" on page 12-10

"Help for Subfunctions and Private Functions" on page 12-10

"Help for Methods and Overloaded Functions" on page 12-10

Using the Help Browser

Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

Some of the features of the Help browser are listed below.

Feature	Description
Product Filter	Establish which products to find help on.
Contents	Look up topics in the Table of Contents.
Index	Look up help using the documentation Index.
Search	Search the documentation for one or more words.
Demos	See what demos are available; run selected demos.
Favorites	Save bookmarks for frequently used Help pages.

For more information: See Finding Information with the Help Browser in the MATLAB Desktop Tools and Development Environment documentation.

Help on Functions from the Help Browser

To find help on any function from the Help browser, do either of the following:

- Select the **Contents** tab of the Help browser, open the **Contents** entry labeled `MATLAB`, and find the two subentries shown below. Use one of these to look up the function you want help on.
  - Functions — Categorical List
  - Functions — Alphabetical List
- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

  ```
  help
  ```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help` *`category`*. For example,

  ```
  help datafun
  ```

- To get help on a particular function, type `help functionname`. For example,

  ```
  help sortrows
  ```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
arith	Arithmetic operators
relop	Relational and logical operators
punct	Special character operators
slash	Arithmetic division operators
paren	Parentheses, braces, and bracket operators
precedence	Operator precedence
datatypes	MATLAB data types, their associated functions, and operators that you can overload
lists	Comma separated lists
strings	Character strings
function_handle	Function handles and the @ operator
debug	Debugging functions
java	Using Java from within MATLAB
fileformats	A list of readable file formats
changeNotification	Windows directory change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text in the MATLAB Desktop Tools and Development Environment documentation.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented with M-files. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subdirectory `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...

"Workspace Browser" on page 12-12

"Using the Find and Replace Utility" on page 12-12

"Commenting Out a Block of Code" on page 12-13

"Creating M-Files from Command History" on page 12-13

"Editing M-Files in EMACS" on page 12-13

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **View > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace in the MATLAB Desktop Tools and Development Environment documentation.

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace utility. Click **View > Current Directory**, and then click the binoculars icon at the top of the **Current Directory** window.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like `win` for `windows`, will not be found if enclosed in quotes.

For more information: See Finding and Replacing a String in the MATLAB Desktop Tools and Development Environment documentation.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

1. Highlight the block of text you would like to comment out.
2. Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Commenting in the MATLAB Desktop Tools and Development Environment documentation.

Creating M-Files from Command History

If there is part of your current MATLAB session that you would like to put into an M-file, this is easily done using the Command History window:

1. Open this window by selecting **View > Command History**.
2. Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
3. Right-click once, and select **Create M-File** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing M-Files in EMACS

If you use Emacs, you can download editing modes for editing M-files with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

```
http://www.mathworks.com/matlabcentral/
```

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger in the MATLAB Desktop Tools and Development Environment documentation.

M-File Functions

In this section...
"M-File Structure" on page 12-14
"Using Lowercase for Function Names" on page 12-14
"Getting a Function's Name and Path" on page 12-15
"What M-Files Does a Function Use?" on page 12-15
"Dependent Functions, Built-Ins, Classes" on page 12-16

M-File Structure

An M-File consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See "Basic Parts of an M-File" on page 4-8 in the MATLAB Programming documentation.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

For M-file functions, case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes your M-files more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the name of an M-file that is currently being executed, use the following function in your M-file code.

```
mfilename
```

To include the path along with the M-file name, use

```
mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What M-Files Does a Function Use?

For a simple display of all M-files referenced by a particular function, follow the steps below:

1. Type `clear functions` to clear all functions from memory (see Note below).
2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.
3. Type `inmem` to display all M-Files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

   ```
   [mfiles, mexfiles] = inmem
   ```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

"Getting the Input and Output Arguments" on page 12-17

"Variable Numbers of Arguments" on page 12-17

"String or Numeric Arguments" on page 12-18

"Passing Arguments in a Structure" on page 12-18

"Passing Arguments in a Cell Array" on page 12-19

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `nargchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
disp(nargchk(2, 4, nargin))          % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))      % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
   y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
      1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number 75 as the string, '75'.

```
isnumeric(75)                      isnumeric 75
ans =                              ans =
     1                                  0
```

For more information: See “Passing Arguments with Command and Function Syntax” on page 4-57 in the MATLAB Programming documentation.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...

"Planning the Program" on page 12-20

"Using Pseudo-Code" on page 12-20

"Selecting the Right Data Structures" on page 12-20

"General Coding Practices" on page 12-21

"Naming a Function Uniquely" on page 12-21

"The Importance of Comments" on page 12-21

"Coding in Steps" on page 12-22

"Making Modifications in Steps" on page 12-22

"Functions with One Calling Function" on page 12-22

"Testing the Final Program" on page 12-22

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what data types and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

For more information: See Data Types in the MATLAB Programming documentation.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in an M-file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics® property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%---------------------------------------------
% This function computes the ... <and so on>
%---------------------------------------------
```

For more information: See Comments in the MATLAB Programming documentation.

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It's much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It's better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you've changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same M-file as the calling function, making it a subfunction.

For more information: See Subfunctions in the MATLAB Programming documentation.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

"The MATLAB Debug Functions" on page 12-23

"More Debug Functions" on page 12-23

"The MATLAB Graphical Debugger" on page 12-24

"A Quick Way to Examine Variables" on page 12-24

"Setting Breakpoints from the Command Line" on page 12-25

"Finding Line Numbers to Set Breakpoints" on page 12-25

"Stopping Execution on an Error or Warning" on page 12-25

"Locating an Error from the Error Message" on page 12-25

"Using Warnings to Help Debug" on page 12-26

"Making Code Execution Visible" on page 12-26

"Debugging Scripts" on page 12-26

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging M-Files in the MATLAB Desktop Tools and Development Environment documentation.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
`echo`	Display function or script code as it executes.
`disp`	Display specified values or messages.
`sprintf`, `fprintf`	Display formatted data of different types.

Function	Description
whos	List variables in the workspace.
size	Show array dimensions.
keyboard	Interrupt program execution and allow input from keyboard.
return	Resume execution following a keyboard interruption.
warning	Display specified warning message.
error	Display specified error message.
lasterr	Return error message that was last issued.
lasterror	Return last error message and related information.
lastwarn	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the `open` function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging M-Files and Using Debugging Features in the MATLAB Desktop Tools and Development Environment documentation.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific M-file line number.
- Break at the beginning of a specific subfunction.
- Break at the first executable line in an M-file.
- Break when a warning, or error, is generated.
- Break if any infinite or `NaN` values are encountered.

For more information: See Setting Breakpoints in the MATLAB Desktop Tools and Development Environment documentation.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of an M-file, also numbering each line. To display delaunay`.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `warning debug` to stop execution on any warning and enter debug mode.

For more information: See Debug, Backtrace, and Verbose Modes in the MATLAB Programming documentation.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the M-file being executed in its editor and places the cursor at the point of error.

For more information: See Types of Errors in the MATLAB Desktop Tools and Development Environment documentation.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on "Warning Control" on page 8-24 in the MATLAB Programming documentation to find out how to selectively enable warnings.

For more information: See the `warning` function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

For more information: See Finding Errors in the MATLAB Desktop Tools and Development Environment documentation.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...
"Rules for Variable Names" on page 12-27
"Making Sure Variable Names Are Valid" on page 12-27
"Do Not Use Function Names for Variables" on page 12-28
"Checking for Reserved Keywords" on page 12-28
"Avoid Using i and j for Variables" on page 12-29
"Avoid Overwriting Variables in Scripts" on page 12-29
"Persistent Variables" on page 12-29
"Protecting Persistent Variables" on page 12-29
"Global Variables" on page 12-30

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

```
N = namelengthmax
N =
    63
```

For more information: See "Naming Variables" on page 3-6 in the MATLAB Programming documentation.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See “Naming Variables” on page 3-6 in the MATLAB Programming documentation.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it’s a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See “Potential Conflict with Function Names” on page 3-7 in the MATLAB Programming documentation.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See "M-File Scripts" on page 4-17 in the MATLAB Programming documentation.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See "Persistent Variables" on page 3-5 in the MATLAB Programming documentation.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the M-file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See "Global Variables" on page 3-3 in the MATLAB Programming documentation.

Strings

In this section...

"Creating Strings with Concatenation" on page 12-31
"Comparing Methods of Concatenation" on page 12-31
"Store Arrays of Strings in a Cell Array" on page 12-32
"Converting Between Strings and Cell Arrays" on page 12-32
"Search and Replace Using Regular Expressions" on page 12-33

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;
s = ['There are ' int2str(numChars) ' characters here']
s = sprintf('There are %d characters here\n', numChars)
```

For more information: See "Creating Character Arrays" on page 2-37 and "Converting from Numeric to String" on page 2-59 in the MATLAB Programming documentation.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function, However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See "Cell Arrays of Strings" on page 2-39 in the MATLAB Programming documentation.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...
              'Phoenix      '];
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};
strcmp(charRecord, cellRecord2)
ans =
     0
     1
     0
```

For more information: See "Converting to a Cell Array of Strings" on page 2-40 and "String Comparisons" on page 2-55 in the MATLAB Programming documentation.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.

For more information: See "Regular Expressions" on page 3-30 in the MATLAB Programming documentation.

Evaluating Expressions

In this section...

"Find Alternatives to Using eval" on page 12-34

"Assigning to a Series of Variables" on page 12-34

"Short-Circuit Logical Operators" on page 12-35

"Changing the Counter Variable within a for Loop" on page 12-35

Find Alternatives to Using eval

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that `eval` statements cannot always be translated into C or C++ code by the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See MATLAB Technical Note 1103, "What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?" at URL `http://www.mathworks.com/support/tech-notes/1100/1103.html`.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
   phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical AND and OR operators (&& and ||) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function myfun unless its M-file exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See “Short-Circuit Operators” on page 3-24 in the MATLAB Programming documentation.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable k in the example below) in the body of a for loop. For example, this loop executes just 10 times, even though k is set back to 1 on each iteration.

```
for k = 1:10
   fprintf('Pass %d\n', k)
   k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...

"Precedence Rules" on page 12-36

"File Precedence" on page 12-37

"Adding a Directory to the Search Path" on page 12-37

"Handles to Functions Not on the Path" on page 12-37

"Making Toolbox File Changes Visible to MATLAB" on page 12-38

"Making Nontoolbox File Changes Visible to MATLAB" on page 12-39

"Change Notification on Windows" on page 12-39

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

1. Variable
2. Subfunction
3. Private function
4. Class constructor
5. Overloaded method
6. M-file in the current directory
7. M-file on the path, or MATLAB built-in function

If you have two or more M-files on the path that have the same name, MATLAB selects the function that has its M-file in the directory closest to the beginning of the path string.

For more information: See "Function Precedence Order" on page 9-73 in the MATLAB Programming documentation.

File Precedence

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the directory, MATLAB selects the file to use according to the following precedence:

1. MEX-file
2. MDL-file (Simulink® model)
3. P-Code file
4. M-file

For more information: See “Multiple Implementation Types” on page 4-55 in the MATLAB Programming documentation.

Adding a Directory to the Search Path

To add a directory to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a directory and all of its subdirectories in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subdirectories to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path in the MATLAB Desktop Tools and Development Environment documentation.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path directory as the functions.

If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

1 Create a script in this off-path directory that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
   fhset = @setItems
   fhsort = @sortItems
   fhdel = @deleteItem
```

2 Run the script from your current directory to create the function handles:

```
run E:/testdir/createFhandles
```

3 You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied directories, M-files (and MEX-files) in the *matlabroot*/`toolbox` directories are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear` *functionname*. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot*/`toolbox` directories. If you add (or remove) files from these directories, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For M-files outside of the toolbox directories, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear` *`functionname`*. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...

"Using break, continue, and return" on page 12-40
"Using switch Versus if" on page 12-41
"MATLAB case Evaluates Strings" on page 12-41
"Multiple Conditions in a case Statement" on page 12-41
"Implicit Break in switch-case" on page 12-41
"Variable Scope in a switch" on page 12-42
"Catching Errors with try-catch" on page 12-42
"Nested try-catch Blocks" on page 12-43
"Forcing an Early Return from a Function" on page 12-43

Using break, continue, and return

It's easy to confuse the `break`, `continue`, and `return` functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
`break`	`for` or `while` loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
`continue`	`for` or `while` loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
`return`	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement `switch-case` statements using `if-elseif` instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need `strcmp` to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between `switch-case` statements in MATLAB and C is that you can specify string values in MATLAB `case` statements, which you cannot do in C.

```
switch(method)
   case 'linear'
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with `switch`. The first case below tests for either a `linear` or `bilinear` method by using a cell array in the case statement.

```
switch(method)
   case {'linear', 'bilinear'}
      disp('Method is linear or bilinear')
   case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each case with a `break` statement, code execution falls through to the following `case`. In MATLAB, `case` statements do not fall

through; only one case may execute. Using break within a case statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if result is 52, only the first disp statement executes, even though the second is also a valid match:

```
switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one case of any switch statement, variables defined within one case are not known in the other cases of that switch statement. The same holds true for if-elseif statements.

In these examples, you get an error when choice equals 2, because x is undefined.

```
  -- SWITCH-CASE --
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      plot(x, sin(x));
end
```

```
  -- IF-ELSEIF --

if choice == 1
   x = -pi:0.01:pi;
elseif choice == 2
   plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a try-catch block that will catch any errors and handle them appropriately.

The example below shows a try-catch block within a function that multiplies two matrices. If a statement in the try segment of the block fails, control passes to the catch segment. In this case, the catch statements check the error message that was issued (returned by lasterr) and respond appropriately.

```
try
   X = A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
end
```

For more information: See "The try-catch Statement" on page 8-17 in the MATLAB Programming documentation.

Nested try-catch Blocks

You can also nest `try-catch` blocks, as shown here. You can use this to attempt to recover from an error caught in the first `try` section:

```
try
   statement1                      % Try to execute statement1
catch
   try
      statement2                   % Attempt to recover from error
   catch
      disp 'Operation failed'      % Handle the error
   end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a `return` statement in the function at the point where you want to exit. For example,

```
if <done>
   return
end
```

Save and Load

In this section...

"Saving Data from the Workspace" on page 12-44

"Loading Data into the Workspace" on page 12-44

"Viewing Variables in a MAT-File" on page 12-45

"Appending to a MAT-File" on page 12-45

"Save and Load on Startup or Quit" on page 12-46

"Saving to an ASCII File" on page 12-46

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a `diary` file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, `...`).

For more information: See Saving the Current Workspace in the MATLAB Desktop Tools and Development Environment documentation, "Using the diary Function to Export Data" on page 6-87, and "Using Low-Level File I/O Functions" on page 6-104.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.

- Read a binary or ASCII file using `load`.
- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, `...`).

For more information: See Loading a Saved Workspace and Importing Data in the MATLAB Development Environment documentation, and "Using Low-Level File I/O Functions" on page 6-104.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];   B = 12.5;   C = rand(4);
save savefile;
A = [6 7 8];
```

```
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to `save` the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See "Exporting Delimited ASCII Data Files" on page 6-86.

Files and Filenames

In this section...

"Naming M-files" on page 12-47
"Naming Other Files" on page 12-47
"Passing Filenames as Arguments" on page 12-48
"Passing Filenames to ASCII Files" on page 12-48
"Determining Filenames at Run-Time" on page 12-48
"Returning the Size of a File" on page 12-48

Naming M-files

M-file names must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed M-file name length (returned by the function `namelengthmax`).

```
N = namelengthmax
N =
    63
```

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for an M-file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as M-files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat           % Command syntax
load('mydata.mat')        % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii              % Command syntax
load('mydata.dat','-ascii')         % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

```
function myfun(datafile)
```

- Prompt for the filename using the `input` function

```
filename = input('Enter name of file:  ', 's');
```

- Browse for the file using the `uigetfile` function

```
[filename, pathname] =
   uigetfile('*.mat', 'Select MAT-file');
```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
  -- METHOD #1 --                    -- METHOD #2 --
s = dir('myfile.dat');            fid = fopen('myfile.dat');
filesize = s.bytes                fseek(fid, 0, 'eof');
                                  filesize = ftell(fid)
                                  fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it's a directory (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...

"File I/O Function Overview" on page 12-50

"Common I/O Functions" on page 12-50

"Readable File Formats" on page 12-51

"Using the Import Wizard" on page 12-51

"Loading Mixed Format Data" on page 12-51

"Reading Files with Different Formats" on page 12-52

"Reading ASCII Data into a Cell Array" on page 12-52

"Interactive Input into Your Program" on page 12-52

For more information and examples on importing and exporting data, see Technical Note 1602:

http://www.mathworks.com/support/tech-notes/1600/1602.html

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online "Functions — Categorical List" reference. In the Help browser **Contents**, select **MATLAB > Functions — Categorical List**, and then click **File I/O**.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textread`, `dlmread`, `dlmwrite`. Functions for I/O to text files with comma-separated values are `csvread`, `csvwrite`.

For more information: See Text Files in the MATLAB "Functions — Categorical List" reference documentation.

Readable File Formats

Type `doc fileformats` to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel files) is to use the MATLAB Import Wizard. Open the Import Wizard with the command, `uiimport filename` or by selecting **File > Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Using the Import Wizard" on page 6-11 in the MATLAB Programming documentation.

Loading Mixed Format Data

To load data that is in mixed formats, use `textread` instead of `load`. The `textread` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
[names, x, y] = textread('mydata.dat', '%s %f %d', 1)
```

returns

```
names =
    'Sally'
x =
  12.34000000000000
y =
    45
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Reading ASCII Data into a Cell Array

A common technique used to read an ASCII data file into a cell array is

```
[a,b,c,d] = textread('data.txt', '%s %s %s %s');
mydata = cellstr([a b c d]);
```

For more information: See the `textread` and `cellstr` function reference pages.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See http://www.mathworks.com/support/solutions/data/1-17VEB.html for a more detailed explanation.

For more information: See Reduced Startup Time with Toolbox Path Caching in the MATLAB Desktop Tools and Development Environment documentation.

Operating System Compatibility

In this section...

"Executing O/S Commands from MATLAB" on page 12-54

"Searching Text with grep" on page 12-54

"Constructing Paths and Filenames" on page 12-54

"Finding the MATLAB Root Directory" on page 12-55

"Temporary Directories and Filenames" on page 12-55

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB `!` operator.

On Windows, you can add an ampersand (`&`) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs in the MATLAB Desktop Tools and Development Environment documentation, and the `system` and `dos` function reference pages.

Searching Text with grep

`grep` is a powerful tool for performing text searches in files on UNIX systems. To `grep` from within MATLAB, precede the command with an exclamation point (`!grep`).

For example, to search for the word `warning`, ignoring case, in all M-files of the current directory, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the `fullfile` function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Directory

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the `general` toolbox directory:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Temporary Directories and Filenames

If you need to locate the directory on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this directory.

To create a new file in this directory, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file directory, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop®
- Stateflow®

For more information: See Demos in the Help Browser in the MATLAB Desktop Tools and Development Environment documentation, and the `demo` function reference page.

For More Information

In this section...

"Current CSSM" on page 12-57

"Archived CSSM" on page 12-57

"MATLAB Technical Support" on page 12-57

"Tech Notes" on page 12-57

"MATLAB Central" on page 12-57

"MATLAB Newsletters (Digest, News & Notes)" on page 12-57

"MATLAB Documentation" on page 12-58

"MATLAB Index of Examples" on page 12-58

Current CSSM

```
http://newsreader.mathworks.com/WebX?14@@/comp.soft-sys.matlab
```

Archived CSSM

```
http://mathforum.org/kb/forum.jspa?forumID=80
```

MATLAB Technical Support

```
http://www.mathworks.com/support/
```

Tech Notes

```
http://www.mathworks.com/support/tech-notes/list_all.html
```

MATLAB Central

```
http://www.mathworks.com/matlabcentral/
```

MATLAB Newsletters (Digest, News & Notes)

```
http://www.mathworks.com/company/newsletters/index.html
```

MATLAB Documentation

http://www.mathworks.com/access/helpdesk/help/helpdesk.html

MATLAB Index of Examples

http://www.mathworks.com/access/helpdesk/help/techdoc/demo_example.shtml

Index

( ) symbol
    for indexing into an array 3-103
    for specifying function input arguments 3-103
[ ] symbol
    for concatenating arrays 3-107
    for constructing an array 3-107
    for specifying function return values 3-107
{ } symbol
    for constructing a cell array 3-100
    for indexing into a cell array 3-100
! symbol
    for entering a shell escape function 3-103
% symbol
    for specifying character conversions 3-104
    for writing single-line comments 3-104
    for writing the H1 help line 4-10
' symbol
    for constructing a character array 3-106
* symbol
    for filename wildcards 3-97
, symbol
    for separating array indices 3-99
    for separating array row elements 3-99
    for separating input or output arguments 3-100
    for separating MATLAB commands 3-100
. symbol
    for defining a structure field 3-101
    for specifying object methods 3-101
: symbol
    for converting to a column vector 3-99
    for generating a numeric sequence 3-98
    for preserving array shape on assignment 3-99
    for specifying an indexing range 3-99
    generating a numeric sequence 1-11
; symbol
    for separating rows of an array 3-105
    for suppressing command output 3-105
@ symbol
    for class directories 3-98
    for constructing function handles 3-97
.( ) symbol
    for creating a dynamic structure field 3-102
%{ and %} symbols
    for writing multiple-line comments 3-104
.. symbol
    for referring to a parent directory 3-101
... symbol
    for continuing a command line 3-101

A

access modes
    HDF4 files 7-59
accuracy of calculations 3-14
addition operator 3-16
aggregation 9-40
and (M-file function equivalent for &) 3-20
anonymous functions 5-3
    changing variables 5-9
    constructing 5-3
    evaluating variables 5-8
    in cell arrays 5-6
    multiple anonymous functions 5-13
    passing a function to quad 5-12
    using space characters in 5-6
    with no input arguments 5-5
answer, assigned to ans 3-14
arguments
    checking number of 4-32
    function 4-10
    memory requirements 11-21
    order in argument list 4-36
    order of outputs 4-34
    parsing 4-36
    passing 4-57
    passing variable number 4-34
    to nested functions 4-47

arithmetic operators 3-16
    overloading 9-32
array headers
    memory requirements 11-20
arrays
    cell array of strings 2-39
    concatenating diagonally 1-47
    copying 11-19
    deleting rows and columns 1-35
    diagonal 1-46
    dimensions
        inverse permutation 1-69
    empty 1-49
    expanding 1-31
    flipping 1-38
    functions
        changing indexing style 1-79
        creating a matrix 1-76
        determining data type 1-77
        finding matrix structure or shape 1-77
        modifying matrix shape 1-76
        multidimensional arrays 1-79
        sorting and shifting 1-78
    functions for diagonals 1-78
    getting dimensions of 1-28
    linear indexing 1-19
    multidimensional 1-56
    numeric
        converting to cell array 2-112
    of strings 2-38
    reshaping 1-36
    rotating 1-38
    shifting 1-41
    sorting column data 1-43
    sorting row data 1-43
    sorting row vectors 1-44
    storage 1-19
    transposing 1-37
ASCII data
    exporting 6-84
    exporting delimited data 6-86
    exporting with `diary` function 6-87
    formats 6-75
    importing 6-75
    importing delimited files 6-79
    importing mixed alphabetic and numeric
        data 6-81
    importing space-delimited data 6-78
    reading formatted text 6-113
    saving 6-86
    specifying delimiter used in file 6-79
    with text headers 6-80
    writing 6-114
assert
    formatting strings 2-42
assignment statements
    building structure arrays with 2-75
    local and global variables 3-10
attributes
    retrieving from HDF4 files 7-60
    writing to an HDF4 file 7-69

B

backtrace mode
    warning control 8-31
base (numeric), converting 2-60
base date 2-67
binary data
    controlling data type of values read 6-108
    using the Import Wizard 6-11
    writing to 6-109
binary from decimal conversion 2-60
blanks
    finding in string arrays 2-57
    removing from strings 2-39
built-in functions 3-109
    forcing a built-in call 3-110
    identifying 3-110

C

C++ and MATLAB OOP 9-8
caching
  MATLAB directory 4-14
callback functions
  creating 10-15
  specifying 10-17
calling context 4-18
calling MATLAB functions
  storing as pseudocode 4-15
canonical class 9-9
case conversion 2-63 to 2-64
`cat` 1-60
CDF. *See* Common Data Format
cdfepoch object
  representing CDF time values 7-6
`cell`
  building nested arrays with 2-110
  preallocating empty arrays with 2-99
cell arrays 2-93
  applying functions to 2-108
  converting to numeric array 2-112
  creating 2-95
    with `cells` function 2-99
  deleting cells 2-106
  deleting dimensions 2-106
  flat 2-110
  functions 2-114
  growing 1-32 1-34
  multidimensional 1-73
  nested 2-110
    building with the `cells` function 2-111
    indexing 2-111
  of strings 2-39
    comparing strings 2-56
    functions 2-41
  of structures 2-113
  organizing data 2-109
  preallocating 2-99 11-8
  replacing comma-separated list with 2-107
  reshaping 2-106
  with anonymous function elements 5-6
`char` data type 6-108
character arrays
  categorizing characters of 2-57
  comparing 2-55
  comparing values on cell arrays 2-56
  conversion 2-59
  converting to cell arrays 2-39
  converting to numeric 2-61
  creating 2-37
  delimiting character 2-58
  evaluating 3-27
  expanding 1-35
  finding a substring 2-58
  functions 2-64
  functions that create 2-63
  functions that modify 2-63
  in cell arrays 2-39
  padding for equal row length 2-39
  removing trailing blanks 2-39
  representation 2-37
  scalar 2-56
  searching and replacing 2-58
  searching or comparing 2-64
  token 2-58
  two-dimensional 2-38
  using relational operators on 2-56
characters
  conversion, in format specification
    string 2-47
  corresponding ASCII values 2-61
  finding in string 2-57
  used as delimiters 6-75
characters and strings 2-37
`class` 9-11
class directories 9-6
classes
  clearing definition 9-6
  constructor method 9-10

debugging 9-6
designing 9-9
java 2-118
methods required by MATLAB 9-9
object-oriented methods 9-2
overview 9-2
classes, matlab
overview 2-117
`clear` 4-52 11-24
clipboard
importing binary data 6-11
closing
files 6-115
colon operator 1-11
for multidimensional array subscripting 1-63
scalar expansion with 1-59
column separators
defined 6-75
comma-separated lists 3-79
assigning output from 3-81
assigning to 3-82
FFT example 3-85
generating from cell array 3-79
generating from structure 3-80
replacing with cell array 2-107
usage 3-83
concatenation 3-84
constructing arrays 3-83
displaying arrays 3-84
function call arguments 3-84
function return values 3-85
command/function duality 4-56
comments
in code 4-12
in scripts and functions 4-8
Common Data Format (CDF)
combining records to improve read performance 7-5
converting CDF epoch values to MATLAB datenum values 7-5
reading CDF files 7-2 to 7-3
reading metadata from CDF files 7-2
representing time values 7-6
speeding up read operations 7-4
writing data to CDF files 7-6
comparing
strings 2-55
complex arrays
memory requirements 11-23
complex conjugate transpose operator 3-16
complex number functions 2-31
complex numbers 2-24
creating 2-24
computational functions
applying to cell arrays 2-108
applying to multidimensional arrays 1-70
applying to structure fields 2-83
in M-file 4-8
`computer` 3-14
computer type 3-14
concatenation 1-8
functions 1-9
of diagonal matrices 1-47
of matrices 1-8
of strings 12-31
of unlike data types 1-13
conditional statements 4-32
constructor methods 9-10
guidelines 9-10
using `class` in 9-11
containment 9-40
`Contents.m` file 4-15
control statements
`break` 3-93
`case` 3-89
`catch` 3-94
conditional control 3-87
`continue` 3-93
`else` 3-87
`elseif` 3-87

error control 3-94
for 3-91
if 3-87
loop control 3-91
otherwise 3-89
program termination 3-95
return 3-95
switch 3-89
try 3-94
while 3-92
conv 2-107
conversion characters in format specification
string 2-47
converter methods 9-22
converting
cases of strings 2-63 to 2-64
dates 2-66
numbers 2-59
numeric to string 2-59
string to numeric 2-61
strings 2-59
converting numeric and string data types 2-65
converting numeric to string 2-59
converting string to numeric 2-61
cos 4-17
cputime
versus tic and toc 11-3
creating
cell array 2-95
multidimensional array 1-58
string array 2-39
strings 2-37
structure array 2-75
timer objects 10-5
cross 1-70
curly braces
to nest cell arrays 2-110

D

data
binary, dependence upon array size and type 6-70
data class hierarchy 9-3
data organization
cell arrays 2-109
multidimensional arrays 1-71
structure arrays 2-85
data types 2-3
cell arrays 2-93
cell arrays of strings 2-39
combining unlike data types 1-13
complex numbers 2-24
dates and times 2-66
determining 2-64
double precision 6-108
floating point 2-14
double-precision 2-14
single-precision 2-15
infinity 2-25
integers 2-6
java classes 2-118
logical 2-33
logicals 2-33
NaN 2-26
numeric 2-6
precision 6-108
reading files 6-108
specifying for input 6-108
structure arrays 2-74
user-defined classes 9-3
date 2-71
date and time functions 2-72
datenum 2-68
dates
base 2-67
conversions 2-68
handling and converting 2-66
number 2-67

string, vector of input 2-69
dates and times 2-66
`datestr` 2-68
`datevec` 2-68
`deblank` 2-39
debugging
errors and warnings 8-34
debugging class methods 9-6
decimal representation
to binary 2-60
to hexadecimal 2-60
delaying program execution
using timers 10-2
deleting
cells from cell array 2-106
fields from structure arrays 2-83
matrix rows and columns 1-35
deleting array elements 1-35
deletion operator 1-35
delimiter in string 2-58
delimiters
defined 6-75
diagonal matrices 1-46
`diary` 6-87
dim argument for cat 1-60
dimensions
deleting 2-106
permuting 1-68
removing singleton 1-67
directories
adding to path 9-7
class 9-6
`Contents.m` file 4-15
help for 4-15
MATLAB
caching 4-14
private functions for 5-35
private methods for 9-5
temporary 6-107
disp 2-90
dispatch type 9-73
`display` method 9-13
examples 9-13
displaying
field names for structure array 2-76
division operators
left division 3-16
matrix left division 3-17
matrix right division 3-16
right division 3-16
double precision 6-108
double-precision matrix 2-6
downloading files 6-117
duality, command/function 4-56
dynamic field names in structure arrays 2-80
dynamic regular expressions 3-57

E

Earth Observing System (EOS) 7-36
editor
accessing 4-13
for creating M-files 4-13
`eig` 1-71
element-by-element organization for structures 2-88
`else`, `elseif` 3-88
empty arrays
and `if` statement 3-88
and relational operators 3-18
and `while` loops 3-93
empty matrices 1-49
`end` 1-21
`end` method 9-20
end of file 6-110
EOS (Earth Observing System)
sources of information 7-36
`eps` 3-14
epsilon 3-14
equal to operator 3-18

error 4-19
    formatting strings 2-42
error handling
    debugging 8-34
escape characters
    in format specification string 2-43
evaluating
    string containing function name 3-28
    string containing MATLAB expression 3-27
examples
    checking number of function arguments 4-33
    container class 9-58
    for 3-91
    function 4-19
    if 3-87
    inheritance 9-41
    M-file for structure array 2-84
    polynomial class 9-26
    script 4-17
    switch 3-90
    vectorization 11-4
    while 3-92
expanding
    character arrays 1-35
    structure arrays 2-75
expanding cell arrays 1-32 1-34
expanding structure arrays 1-32 1-34
exporting
    ASCII data 6-84
    in HDF4 format 7-57
    in HDF5 format 7-16
exporting files
    overview 6-3
expressions
    involving empty arrays 3-18
    most recent answer 3-14
    overloading 9-23
    scalar expansion with 3-17
external program, running from MATLAB 3-28

F

`fclose` 6-115
`feof` 6-109
`fid`. *See* file identifiers
field names
    dynamic 2-80
`fieldnames` 2-76
fields 2-74 to 2-76
    accessing data within 2-78
    adding to structure array 2-82
    applying functions to 2-83
        all like-named fields 2-83
    assigning data to 2-75
    deleting from structures 2-83
    indexing within 2-80
    names 2-76
    size 2-81
    writing M-files for 2-84
file exchange
    over Internet 6-117
file I/O
    audio/video files 6-4 6-93
        exporting 6-95
        importing 6-94
    binary files 6-4
    files from the Internet 6-5
    graphics files 6-4 6-90
        exporting 6-91
        importing 6-91
    internet 6-117
        downloading from web 6-117
        FTP operations 6-122
        sending e-mail 6-120
        ZIP files 6-119
    low-level functions 6-104
        ASCII files:exporting 6-114
        ASCII files:importing 6-113
        binary files:exporting 6-109
        binary files:importing 6-107

MAT-files
exporting 6-64
MATLAB HDF4 utility API 7-71
memory mapping. *See* memory mapping
overview 6-3
Import Wizard 6-5
large data sets 6-6
low-level functions 6-6
toolboxes for importing data 6-7
scientific formats 7-1
CDF files 7-2
FITS files 7-8
HDF4 and HDF-DOS files 7-53
HDF4 files 7-36 7-57
HDF5 files 7-11
spreadsheet files 6-5 6-98
Lotus 123 6-101
Microsoft Excel 6-98
supported file formats 6-9
supported file types 6-3
system clipboard 6-5
text files
exporting 6-84
importing 6-75
text files 6-4
using Import Wizard 6-11
file identifiers
clearing 6-116
defined 6-105
file import and export
overview 6-3
supported file types 6-3
file operations
FTP 6-122
file types
audio, video 6-4
binary 6-4
graphics 6-4
spreadsheets 6-5
supported by MATLAB 6-3
text 6-4
filenames
wildcards 3-97
files
ASCII
reading 6-112
reading formatted text 6-113
writing 6-114
beginning of 6-110
binary
controlling data type values read 6-108
data types 6-108
reading 6-107
writing to 6-109
closing 6-115
current position 6-110
end of 6-110
failing to open 6-106
file identifiers (FID) 6-105
MAT 6-73
opening 6-105
permissions 6-105
position 6-109
specifying delimiter used in ASCII files 6-79
temporary 6-107
`find` function
and subscripting 3-22
finding
substring within a string 2-58
FITS. *See* Flexible Image Transport System
Flexible Image Transport System (FITS)
reading 7-8
reading data 7-9
reading metadata 7-8
flipping matrices 1-38
float 6-108
floating point 2-14
floating point, double-precision 2-14
converting to 2-16
creating 2-15

maximum and minimum values 2-18
floating point, single-precision 2-15
converting to 2-16
creating 2-16
maximum and minimum values 2-18
floating-point functions 2-30
floating-point numbers
largest 3-14
smallest 3-14
floating-point precision 6-108
floating-point relative accuracy 3-14
flow control
`break` 3-93
`case` 3-89
`catch` 3-94
conditional control 3-87
`continue` 3-93
`else` 3-87
`elseif` 3-87
error control 3-94
`for` 3-91
`if` 3-87
loop control 3-91
`otherwise` 3-89
program termination 3-95
`return` 3-95
`switch` 3-89
`try` 3-94
`while` 3-92
`fopen` 6-105
failing 6-106
`for` 2-112
example 3-91
indexing 3-92
nested 3-91
syntax 3-91
format for numeric values 2-27
formatting strings 2-42
field width 2-49
flags 2-50
format operator 2-45
precision 2-48
setting field width 2-51 to 2-52
setting precision 2-51 to 2-52
subtype 2-48
using identifiers 2-53
value identifiers 2-51
fprintf
formatting strings 2-42
`fread` 6-107
`frewind` 6-109
`fseek` 6-109
`ftell` 6-109
FTP file operations 6-122
function calls
memory requirements 11-21
function definition line
for subfunction 5-33
in an M-file 4-8
syntax 4-9
function handles
example 4-24
for nested functions 5-21
maximum name length 4-30
naming 4-30
operations on 4-25
overview 2-115
overview of 4-22
function types
overloaded 5-37
function workspace 4-18
functions
applying
to multidimensional structure arrays 1-75
to structure contents 2-83
applying to cell arrays 2-108
arguments
passing variable number of 4-34
body 4-8 4-11

built-in 3-109
    forcing a built-in call 3-110
    identifying 3-110
calling
    command syntax 4-56
    function syntax 4-57
    passing arguments 4-57
calling context 4-18
cell arrays 2-114
cell arrays of strings 2-41
changing indexing style 1-79
character arrays 2-64
clearing from memory 4-52
comments 4-8
comparing character arrays 2-64
complex number 2-31
computational, applying to structure fields 2-83
creating a matrix 1-76
creating arrays with 1-60
creating matrices 1-5
date and time 2-72
determining data type 1-77
example 4-19
executing function name string 3-28
finding matrix structure or shape 1-77
floating-point 2-30
for diagonal matrices 1-78
infinity 2-31
integer 2-30
logical array 2-34
M-file 3-108
matrix concatenation 1-9
modifying character arrays 2-63
modifying matrix shape 1-76
multidimensional arrays 1-79
multiple output arguments 4-10
naming
    conflict with variable names 3-7
NaN 2-31
numeric and string conversion 2-65
numeric to string conversion 2-59
output formatting 2-32
overloaded 3-110
overloading 9-25
primary 5-33
searching character arrays 2-64
sorting and shifting 1-78
sparse matrix 1-54
storing as pseudocode 4-15
string to numeric conversion 2-61
structures 2-92
that determine data type 2-64
type identification 2-32
types of 4-19
    anonymous 5-3
    nested 5-16
    overloaded 5-37
    primary 5-15
    private 5-35
    subfunctions 5-33
`fwrite` 6-109

G

`get` method 9-14
global attributes
    HDF4 files 7-60
global variables 3-3
    alternatives 3-5
    creating 3-4
    displaying 3-4
    suggestions for use 3-4
graphics files
    getting information about 6-90
    importing and exporting 6-90
greater than operator 3-18
greater than or equal to operator 3-18
growing an array 1-31
growing cell array 1-32 1-34

growing structure arrays 1-32 1-34

H

H1 line 4-8 4-10
  and `help` command 4-8
  and `lookfor` command 4-8
HDF Import Tool
  using 7-36
  using subsetting options 7-41
HDF-EOS
  Earth Observing System 7-36
HDF4 7-36
  closing a data set 7-70
  closing a file 7-71
  closing all open identifiers 7-72
  closing data sets 7-63
  creating a file 7-65
  creating data sets 7-65
  exporting in HDF4 format 7-57
  importing data 7-54
  importing subsets of data 7-39
  listing all open identifiers 7-71
  low-level functions
    overview 7-56
    reading data 7-58
  mapping HDF4 syntax to MATLAB syntax 7-57
  MATLAB utility API 7-71
  opening files 7-59
  overview 7-36
  reading data 7-62
  reading data set metadata 7-61
  reading data sets 7-61
  reading global attributes 7-60
  reading metadata 7-59
  selecting data sets to import 7-38
  specifying file access modes 7-59
  using `hdfinfo` to import metadata 7-53
  using high-level functions
    overview 7-53
  using predefined attributes 7-69
  using the HDF Import Tool 7-36
  writing data 7-64 7-67
  writing metadata 7-69
  See also HDF5
HDF5 7-11
  exporting data in HDF5 format 7-16
  low-level functions
    mapping HDF5 data types to MATLAB data types 7-29
    mapping HDF5 syntax to MATLAB syntax 7-27
    reading and writing data 7-31
  overview 7-11
  using `hdf5info` to read metadata 7-11
  using `hdf5read` to import data 7-15
  using high-level functions 7-11
  using low-level functions 7-26
  See also HDF4
`help`
  and H1 line 4-8
  M-file 4-11
help text 4-8
hexadecimal, converting from decimal 2-60
Hierarchical Data Format. *See* HDF4. *See* HDF5
hierarchy of data classes 9-3
hyperthreading 11-16

I

`if`
  and empty arrays 3-88
  example 3-87
  nested 3-88
imaginary unit 3-14
Import Data option 6-11
import functions
  comparison of features 6-77

Import Wizard
importing binary data 6-11
overview 6-5
importing
ASCII data 6-75
HDF4 data 7-53
from the command line 7-56
selecting HDF4 data sets 7-38
subsets of HDF4 data 7-39
importing files
overview 6-3
indexed reference 9-15
indexing
`for` loops 3-92
multidimensional arrays 1-62
nested cell arrays 2-111
nested structure arrays 2-91
structures within cell arrays 2-113
within structure fields 2-80
indices, how MATLAB calculates 1-65
`Inf` 3-14
`inferiorto` 9-71
`inferiorto` function 9-71
infinity 2-25
functions 2-31
represented in MATLAB 3-14
inheritance
example class 9-41
multiple 9-40
simple 9-38
inputParser class
arguments that default 4-43
building the schema 4-38
case-sensitive matching 4-45
constructor 4-38
defined 4-36
handling unmatched arguments 4-44
method summary 4-46
parsing parameters 4-40
passing arguments in a structure 4-41
property summary 4-46
integer data type 6-114
integer functions 2-30
integers 2-6
creating 2-7
largest system can represent 3-14
smallest system can represent 3-14
Internet functions 6-117
`intmax` 3-14
`intmin` 3-14
inverse permutation of array dimensions 1-69
`ipermute` 1-69
isa 9-12

J

Java and MATLAB OOP 9-8
java classes 2-118

K

keywords 3-13
checking for 12-28

L

large data sets
memory usage in array storage 11-19
memory usage in function calls 11-22
less than operator 3-17
less than or equal to operator 3-17
`load` 11-24
loading objects 9-64
`loadobj` example 9-66
local variables 3-2
logical array functions 2-34
`logical` data type 2-33
logical expressions
and subscripting 3-22
logical operators 3-19
bit-wise 3-23

elementwise 3-19
short-circuit 3-24
logical types 2-33
`long` 6-108
long integer 6-108
`lookfor` 4-8 4-10
and H1 line 4-8
loops
`for` 3-91
`while` 3-92

M

M-file functions
identifying 3-108
M-files
comments 4-12
contents 4-8
corresponding to functions 9-23
creating
in MATLAB directory 4-14
creating with text editor 4-13
kinds 4-7
naming 4-7
operating on structures 2-84
overview 4-8
primary function 5-15
subfunction 5-33
superseding existing names 5-34
mapping memory. *See* memory mapping
MATLAB
data type classes 9-3
programming
M-files 4-7
scripts 4-17
structures 9-7
version 3-14
matrices
accessing multiple elements 1-20
accessing single elements 1-18
concatenating 1-8
concatenating diagonally 1-47
constructing a matrix operations
constructing 1-4
creating 1-3
data structure query 1-30
data type query 1-29
deleting rows and columns 1-35
diagonal 1-46
double-precision 2-6
empty 1-49
expanding 1-31
flipping 1-38
`for` loop index 3-92
functions
changing indexing style 1-79
creating a matrix 1-76
determining data type 1-77
finding matrix structure or shape 1-77
modifying matrix shape 1-76
sorting and shifting 1-78
functions for creating 1-5
functions for diagonals 1-78
getting dimensions of 1-28
linear indexing 1-19
reshaping 1-36
rotating 1-38
scalar 1-51
See also matrices 3-92
shifting 1-41
single-precision 2-6
sorting column data 1-43
sorting row data 1-43
sorting row vectors 1-44
transposing 1-37
vectors 1-52
matrix operations
concatenating matrices 1-8
creating matrices 1-3
`mean` 1-70

memory
- function workspace 4-18
- making efficient use of 11-18
- management 11-24
- `Out of Memory` message 11-27

memory mapping
- demonstration 6-58
- `memmapfile` class
  - class constructor 6-29
  - class methods 6-56
  - class properties 6-27
  - defined 6-27
  - `Filename` property 6-32
  - `Format` property 6-34
  - `Offset` property 6-34
  - `Repeat` property 6-41
  - supported formats 6-40
  - `Writable` property 6-42
- overview 6-23
  - benefits of 6-24
  - byte ordering 6-26
  - limitations of 6-25
  - when to use 6-26
- reading from file 6-43
- removing map 6-58
- selecting file to map 6-32
- setting access privileges 6-42
- setting extent of map 6-41
- setting start of map 6-34
- specifying data types in file 6-34
- supported data types 6-40
- writing to file 6-48

memory requirements
- array headers 11-20
- for array allocation 11-18
- for complex arrays 11-23
- for copying arrays 11-19
- for creating and modifying arrays 11-18
- for handling variables in 11-18
- for numeric arrays 11-22
- for passing arguments 11-21
- for sparse matrices 11-23

message identifiers
- using with warnings 8-26

methods 9-2
- converters 9-22
- determining which is called 4-55
- `display` 9-13
- `end` 9-20
- `get` 9-14
- invoking on objects 9-4
- listing 9-36
- precedence 9-72
- required by MATLAB 9-9
- `set` 9-14
- `subsasgn` 9-15
- `subsref` 9-15

multidimensional arrays
- applying functions 1-70
  - element-by-element functions 1-70
  - matrix functions 1-70
  - vector functions 1-70
- cell arrays 1-73
- computations on 1-70
- creating 1-58
  - at the command line 1-58
  - with functions 1-60
  - with the `cat` function 1-60
- extending 1-59
- format 1-62
- indexing 1-62
  - avoiding ambiguity 1-66
  - with the colon operator 1-63
- number of dimensions 1-62
- organizing data 1-71
- permuting dimensions 1-68
- removing singleton dimensions 1-67
- reshaping 1-66
- size of 1-62
- storage 1-62

structure arrays 1-74
applying functions 1-75
subscripts 1-57
multiple conditions for `switch` 3-90
multiple inheritance 9-40
multiplication operators
matrix multiplication 3-16
multiplication 3-16
multithreaded computation 11-14

N

names
structure fields 2-76
superseding 5-34
NaN 2-26 3-14
functions 2-31
logical operations on 2-26
`nargin` 4-32
checking input arguments 4-32
in nested functions 4-47
`nargout` 4-32
checking output arguments 4-32
in nested functions 4-47
`ndgrid` 1-79
`ndims` 1-62
nested functions 5-16
creating 5-16
example — creating a function handle 5-27
example — function-generating functions 5-29
passing optional arguments 4-47
separate variable instances 5-25
using function handles with 5-21
variable scope in 5-19
nesting
cell arrays 2-110
for loops 3-91
`if` statements 3-88
structures 2-91
newlines in string arrays 2-57
`not` (M-file function equivalent for ~) 3-20
not a number (NaN) 2-26
not equal to operator 3-18
Not-a-Number 3-14
`now` 2-71
number of arguments 4-32
numbers
date 2-67
time 2-67
numeric arrays
memory requirements 11-22
numeric data types 2-6
conversion functions 2-65
converting to strings 2-59
setting display format 2-27
numeric to string conversion functions 2-59

O

object-oriented programming
features of 9-3
inheritance
multiple 9-40
simple 9-38
overloading 9-23
subscripting 9-16
See also classes and objects 9-12
objects
accessing data in 9-13
as indices into objects 9-21
creating 9-4
invoking methods on 9-4
loading 9-64
overview 9-2
precedence 9-70
saving 9-64
offsets for indexing 1-65
online help 4-11

opening
  files
    failing 6-106
    HDF4 files 7-59
    permissions 6-105
    using low-level functions 6-105
operator precedence 3-25
  overriding 3-26
operators
  addition 3-16
  applying to cell arrays 2-108
  applying to structure fields 2-83
  arithmetic 3-16
  categories 3-16
  colon 3-16
  complex conjugate transpose 3-16
  deletion 1-35
  equal to 3-18
  greater than 3-18
  greater than or equal to 3-18
  left division 3-16
  less than 3-17
  less than or equal to 3-17
  logical 3-19
    bit-wise 3-23
    elementwise 3-19
    short-circuit 3-24
  matrix left division 3-17
  matrix multiplication 3-16
  matrix power 3-17
  matrix right division 3-16
  multiplication 3-16
  not equal to 3-18
  overloading 9-3
  power 3-16
  relational 3-17
  right division 3-16
  subtraction 3-16
  table of 9-23
  transpose 3-16
  unary minus 3-16
  unary plus 3-16
optimization
  preallocation, array 11-7 11-25
  vectorization 11-4
`or` (M-file function equivalent for `|`) 3-20
organizing data
  cell arrays 2-109
  multidimensional arrays 1-71
  structure arrays 2-85
`Out of Memory` message 11-27
output arguments 4-10
  order of 4-34
output formatting functions 2-32
overloaded functions 3-110 5-37
overloading 9-16
  arithmetic operators 9-32
  functions 9-25
  loadobj 9-65
  operators 9-3
  `pie3` 9-61
  `saveobj` 9-65

P

`pack` 11-24
page subscripts 1-57
parentheses
  for input arguments 4-10
  overriding operator precedence with 3-26
parsing input arguments 4-36
Paste Special option 6-11
path
  adding directories to 9-7
`pcode` 4-15
percent sign (comments) 4-12
performance
  analyzing 11-2
permission strings 6-105
`permute` 1-68

permuting array dimensions 1-68
inverse 1-69
persistent variables 3-5
initializing 3-6
pi 3-14
pie3 function overloaded 9-61
plane organization for structures 2-87
polar 4-18
polynomials
example class 9-26
power operators
matrix power 3-17
power 3-16
preallocation
arrays 11-7 11-25
cell array 11-8
precedence
object 9-70
operator 3-25
overriding 3-26
precision
char 6-108
data types 6-108
double 6-108
float 6-108
long 6-108
short 6-108
single 6-108
uchar 6-108
primary functions 5-15
private directory 5-35
private functions 5-35
precedence of in classes 9-74
precedence of when calling 4-54
private methods 9-5
program control
break 3-93
case 3-89
catch 3-94
conditional control 3-87
continue 3-93
else 3-87
elseif 3-87
error control 3-94
for 3-91
if 3-87
loop control 3-91
otherwise 3-89
program termination 3-95
return 3-95
switch 3-89
try 3-94
while 3-92
programs
running external 3-28
pseudocode 4-15 to 4-16

Q

quit 11-24

R

randn 1-60
reading
HDF4 data 7-53
from the command line 7-56
selecting HDF4 data sets 7-38
subsets of HDF4 data 7-39
realmax 3-14
realmin 3-14
reference, subscripted 9-16
regexp 3-31
regexpi 3-31
regexprep 3-31
regexptranslate 3-31
regular expression metacharacters

- character classes
  - match alphanumeric character (\w) 3-35
  - match any character (period) 3-34
  - match any characters but these ([^c1c2c3]) 3-33
  - match any of these characters ([c1c2c3]) 3-34
  - match characters in this range ([c1-c2]) 3-35
  - match digit character (\d) 3-35
  - match nonalphanumeric character (\W) 3-33
  - match nondigit character (\D) 3-33
  - match nonwhitespace character (\S) 3-33
  - match whitespace character (\s) 3-35
- character representation
  - alarm character (\a) 3-36
  - backslash character (\\) 3-36 3-73
  - backspace character (\b) 3-36
  - carriage return character (\r) 3-36
  - dollar sign (\$) 3-36 3-73
  - form feed character (\f) 3-36
  - hexadecimal character (\x) 3-36
  - horizontal tab character (\t) 3-36
  - literal character (\char) 3-36
  - new line character (\n) 3-36
  - octal character (\o) 3-36
  - vertical tab character (\v) 3-36
- conditional operators
  - if condition, match expr ((?(condition)expr)) 3-55 3-77
- dynamic expressions
  - pattern matching functions 3-61
  - pattern matching scripts 3-62
  - replacement expressions 3-60
  - string replacement functions 3-64
- logical operators
  - atomic group ((?>expr)) 3-37
  - comment (?#expr) 3-39
  - grouping and capture (expr) 3-37
  - grouping only (?:expr) 3-37
  - match exact word (\<expr\>) 3-40
  - match expr1 or expr2 (expr1|expr2) 3-38
  - match if expression begins string (^expr) 3-39
  - match if expression begins word (\<expr) 3-40
  - match if expression ends string (expr$) 3-39
  - match if expression ends word (expr\>) 3-40
  - noncapturing group ((?:expr)) 3-37
- lookaround operators
  - match expr1, if followed by expr2 (expr1(?=expr2)) 3-42
  - match expr1, if not followed by expr2 (expr1(?!expr2)) 3-42
  - match expr2, if not preceded by expr1 (expr1(?<!expr2)) 3-44
  - match expr2, if preceded by expr1 (expr1(?<=expr2)) 3-43
- operator summary 3-71
- quantifiers
  - lazy quantifier (quant?) 3-47
  - match 0 or 1 instance (expr?) 3-46
  - match 0 or more instances (expr*) 3-46
  - match 1 or more instances (expr+) 3-47
  - match at least m instances (expr{m,}) 3-45
  - match m to n instances (expr{m,n}) 3-47
  - match n instances (expr{n}) 3-45

- token operators
    - conditional with named token ((?(name)s1|s2)) 3-53
    - create named token ((?<name>expr)) 3-53
    - create unnamed token ((expr)) 3-48
    - give name to token ((?<name>expr))) 3-53
    - if token, match expr1, else expr2 ((?(token)expr1|expr2)) 3-55
    - match named token (\k<name>) 3-53
    - match Nth token (\N) 3-48
    - replace Nth token ($N) 3-49
    - replace Nth token (N) 3-49
    - replace with named token (?<name>) 3-53

regular expressions
- character classes 3-33
- character representation 3-36
- conditional expressions 3-55
- dynamic expressions 3-57
    - example 3-58
- functions
    - `regexp` 3-31
    - `regexpi` 3-31
    - `regexprep` 3-31
    - `regexptranslate` 3-31
- introduction 3-30
- logical operators 3-37
- lookaround operators 3-39
    - used in logical statements 3-44
- multiple strings
    - finding a single pattern 3-68
    - finding multiple patterns 3-70
    - matching 3-68
    - replacing 3-70
- quantifiers 3-45
    - lazy 3-47
- tokens 3-48
    - example 1 3-50
    - example 2 3-50
    - introduction 3-49
    - named capture 3-53
    - operators 3-48
    - use in replacement string 3-53

relational operators 3-17
- empty arrays 3-18
- strings 2-56

removing
- cells from cell array 2-106
- fields from structure arrays 2-83
- singleton dimensions 1-67

replacing substring within string 2-58

`repmap` 1-60

`reshape` 1-66 2-106

reshaping
- cell arrays 2-106
- multidimensional arrays 1-66

reshaping matrices 1-36

`rmfield` 2-83

rotating matrices 1-38

S

`save` 11-24

`saveobj` example 9-66

saving
- objects 9-64

scalar
- and relational operators 2-56
- expansion 3-17
- string 2-56

scalars 1-51

scheduling program execution
- using timers 10-2

scripts 4-7
- example 4-17
- executing 4-18

search path
  M-files on 5-34
set method 9-14
shell escape functions 3-28
shiftdim 1-79
shifting matrix elements 1-41
short 6-108
short integer 6-108
short-circuiting
  in conditional expressions 3-21
  operators 3-24
simple inheritance 9-38
sin 1-70
single precision 6-108
single-precision matrix 2-6
size 2-81
  structure arrays 2-81
  structure fields 2-81
smallest value system can represent 3-14
sorting matrix column data 1-43
sorting matrix row data 1-43
sorting matrix row vectors 1-44
(space) character
  for separating array row elements 3-106
  for separating function return values 3-106
sparse matrices
  memory requirements 11-23
sparse matrix functions 1-54
sprintf 6-115
  formatting strings 2-42
square brackets
  for output arguments 4-10
squeeze 1-67
  with multidimensional arguments 1-71
sscanf 6-114
starting
  timers 10-10
statements
  conditional 4-32
stopping
  timers 10-10
strcmp 2-55
string to numeric conversion
  functions 2-61
strings 2-37
  comparing 2-55
  converting to numeric 2-61
  functions to create 2-63
  searching and replacing 2-58
strings, cell arrays of 2-39
strings, formatting 2-42
  escape characters 2-43
  field width 2-49
  flags 2-50
  format operator 2-45
  precision 2-48
  setting field width 2-51 to 2-52
  setting precision 2-51 to 2-52
  subtype 2-48
  using identifiers 2-53
  value identifiers 2-51
structs 2-76
  for nested structures 2-91
structure arrays 2-74
  accessing data 2-78
  adding fields to 2-82
  applying functions to 2-83
  building 2-75
    using structs 2-76
  data organization 2-85
  deleting fields 2-83
  dynamic field names 2-80
  element-by-element organization 2-88
  expanding 2-75
  fields 2-74
    assigning data to 2-75
  growing 1-32 1-34
  indexing
    nested structures 2-91
    within fields 2-80

multidimensional 1-74
applying functions 1-75
nesting 2-91
obtaining field names 2-76
organizing data 2-85
example 2-89
plane organization 2-87
size 2-81
subarrays, accessing 2-79
subscripting 2-75
used with classes 9-7
within cell arrays 2-113
writing M-files for 2-84
example 2-84
structures
field names
dynamic 2-80
functions 2-92
subfunctions 5-33
accessing 5-34
creating 5-33
debugging 5-34
definition line 5-33
precedence of 4-54
subsasgn
for index reference 9-15
for subscripted assignment 9-19
subscripted assignment 9-19
subscripting
how MATLAB calculates indices 1-65
multidimensional arrays 1-57
overloading 9-16
page 1-57
structure arrays 2-75
with logical expression 3-22
with the find function 3-22
subsref 9-16
subsref method 9-15
substring within a string 2-58
subtraction operator 3-16
sum 1-70
superiorto 9-71
superseding existing M-file names 5-34
switch
case groupings 3-89
example 3-90
multiple conditions 3-90
symbols 3-96
asterisk * 3-96
at sign @ 3-97
colon : 3-98
comma , 3-99
curly braces { } 3-100
dot . 3-100
dot-dot .. 3-101
dot-dot-dot ... 3-101
dot-parentheses .( ) 3-102
exclamation point ! 3-103
parentheses ( ) 3-103
percent % 3-103
percent-brace %{ and %} 3-104
semicolon ; 3-104
single quotes ' 3-105
space character 3-106
square brackets [ ] 3-107

T

tabs in string arrays 2-57
tempdir 6-107
tempname 6-107
temporary files
creating 6-107
text files
reading 6-112
tic and toc
versus cputime 11-3
time
numbers 2-67
time and date functions 2-72

timer objects
  blocking the command line 10-12
  callback functions 10-14
  creating 10-5
  deleting 10-5
  execution modes 10-19
  finding all existing timers 10-24
  naming convention 10-6
  overview 10-2
  properties 10-7
  starting 10-10
  stopping 10-10
timers
  starting and stopping 10-10
  using 10-2
times and dates 2-66
tips, programming
  additional information 12-57
  command and function syntax 12-4
  debugging 12-23
  demos 12-56
  development environment 12-12
  evaluating expressions 12-34
  files and filenames 12-47
  function arguments 12-17
  help 12-7
  input/output 12-50
  M-file functions 12-14
  MATLAB path 12-36
  operating system compatibility 12-54
  program control 12-40
  program development 12-20
  save and load 12-44
  starting MATLAB 12-53
  strings 12-31
  variables 12-27
token in string 2-58
tokens
  regular expressions 3-48
tolerance 3-14
`transpose` 1-69
transpose operator 3-16
transposing matrices 1-37
trigonometric functions 1-70
type identification functions 2-32

U

`uchar` data type 6-108
unary minus operator 3-16
unary plus operator 3-16
user classes, designing 9-9

V

value
  data type 6-108
  largest system can represent 3-14
`varargin` 2-108 4-35
  in argument list 4-36
  in nested functions 4-47
  unpacking contents 4-35
`varargout` 4-35
  in argument list 4-36
  in nested functions 4-47
  packing contents 4-35
variables
  global 3-3
    alternatives 3-5
    creating 3-4
    displaying 3-4
    recommendations 3-11
    suggestions for use 3-4
  in evaluation statements 3-9
  lifetime of 3-12
  loaded from a MAT-file 3-8
  local 3-2
  naming 3-6
    conflict with function names 3-7

persistent 3-5
initializing 3-6
replacing list with a cell array 2-107
scope 3-10
in nested functions 3-12
storage in memory 11-18
usage guidelines 3-10
vector
of dates 2-69
preallocation 11-7 11-25
vectorization 11-4
example 11-4
replacing `for`
vectorization 3-91
vectors 1-52
verbose mode
warning control 8-31
`version` 3-14
obtaining 3-14

W

warning
formatting strings 2-42
warning control 8-24
backtrace, verbose modes 8-31
saving and restoring state 8-30
warning control statements
message identifiers 8-26
output from 8-28
output structure array 8-29
warnings
debugging 8-34
identifying 8-23
syntax 8-25
warning control statements 8-26
warning states 8-26
Web content access 6-117
`which` 4-55
used with methods 9-75
`while`
empty arrays 3-93
example 3-92
syntax 3-92
white space
finding in string 2-57
`whos` 1-62
interpreting memory use 11-24
wildcards, in filenames 3-97
workspace
context 4-18
of individual functions 4-18
writing
ASCII data 6-84
HDF4 data 7-67
in HDF4 format 7-57
in HDF5 format 7-16

Z

`zeros` 1-60