How to Contact The MathWorks:

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

http://www.mathworks.com Web
ftp.mathworks.com Anonymous FTP server
comp.soft-sys.matlab Newsgroup

support@mathworks.com Technical support
suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
subscribe@mathworks.com Subscribing user registration
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information

Using MATLAB
© COPYRIGHT 1984 - 1997 by The MathWorks, Inc. All Rights Reserved.
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.

U.S. GOVERNMENT: If Licensee is acquiring the software on behalf of any unit or agency of the U.S. Government, the following shall apply:

(a) for units of the Department of Defense:
RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.
(b) for any other unit or agency:
NOTICE - Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Clause 52.227-19(c)(2) of the FAR.

Contractor/manufacturer is The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760-1500.
MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.
Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: December 1996 First printing For MATLAB 5.0
May 1997 Revised for 5.1 (online version)
Contents

1 Introduction

What Is MATLAB? ............................................. 1-3
The MATLAB System .................................... 1-5
How to Use the Documentation Set ................... 1-7
About Simulink ............................................. 1-9
About Toolboxes ........................................... 1-10

2 Using the Environment

Starting MATLAB ........................................ 2-2
The Command Window .................................. 2-3
Editing M-Files ........................................... 2-8
Preference Setting ....................................... 2-13
Command History ....................................... 2-17
The MATLAB Workspace ............................... 2-18
The MATLAB Search Path ............................. 2-26
Help and Online Documentation ....................... 2-31
Disk File Manipulation and Shell Escape ............. 2-35
Data Import/Export ..................................... 2-36
The Startup M-File ..................................... 2-41
Memory Utilization ................................................. 2-42
License Manager Administration (UNIX) ................. 2-44

Debugger and Profiler

MATLAB Debugger ............................................. 3-2
M-File Profiler .................................................. 3-21

Matrices and Linear Algebra

Matrices and Linear Algebra ................................. 4-2
Matrices in MATLAB .......................................... 4-4
Solving Linear Equations .................................... 4-14
Inverses and Determinants .................................... 4-22
LU, QR, and Cholesky Factorizations ...................... 4-26
Matrix Powers and Exponentials ............................ 4-33
Eigenvalues ...................................................... 4-36
Singular Value Decomposition .............................. 4-40
Polynomials and Interpolation

Polynomials ............................... 5-2
Interpolation ............................... 5-9

Data Analysis and Statistics

Column-Oriented Data Sets .................. 6-3
Basic Data Analysis Functions .............. 6-7
Data Pre-Processing ......................... 6-12
Regression and Curve Fitting ............... 6-15
Case Study: Curve Fitting ................. 6-20
Difference Equations and Filtering .......... 6-29
Fourier Analysis and the Fast Fourier Transform (FFT) . 6-31

Function Functions

Representing Functions in MATLAB .......... 7-3
Plotting Mathematical Functions .......... 7-4
Minimizing Functions and Finding Zeros .... 7-7
Numerical Integration (Quadrature) ........ 7-14
Ordinary Differential Equations

8

Quick Start ........................................ 8-3
Representing Problems ............................ 8-5
ODE Solvers ....................................... 8-10
Creating ODE Files ................................. 8-14
Improving Solver Performance .................... 8-17
Examples: Applying the ODE Solvers ............. 8-33
Questions and Answers ............................. 8-47

Sparse Matrices

9

Introduction ..................................... 9-5
Viewing Sparse Matrices ......................... 9-11
Example: Adjacency Matrices and Graphs ....... 9-15
Sparse Matrix Operations ......................... 9-23

M-File Programming

10

MATLAB Programming: A Quick Start .......... 10-2
Scripts ............................................. 10-5
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local and Global Variables</td>
<td>10-16</td>
</tr>
<tr>
<td>Data Types</td>
<td>10-19</td>
</tr>
<tr>
<td>Operators</td>
<td>10-22</td>
</tr>
<tr>
<td>Flow Control</td>
<td>10-29</td>
</tr>
<tr>
<td>Subfunctions</td>
<td>10-35</td>
</tr>
<tr>
<td>Indexing and Subscripting</td>
<td>10-37</td>
</tr>
<tr>
<td>String Evaluation</td>
<td>10-43</td>
</tr>
<tr>
<td>Command/Function Duality</td>
<td>10-45</td>
</tr>
<tr>
<td>Empty Matrices</td>
<td>10-46</td>
</tr>
<tr>
<td>Errors and Warnings</td>
<td>10-48</td>
</tr>
<tr>
<td>Times and Dates</td>
<td>10-51</td>
</tr>
<tr>
<td>Obtaining User Input</td>
<td>10-57</td>
</tr>
<tr>
<td>Shell Escape Functions</td>
<td>10-58</td>
</tr>
<tr>
<td>Optimizing the Performance of MATLAB Code</td>
<td>10-59</td>
</tr>
</tbody>
</table>

## 11 Character Arrays (Strings)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Arrays</td>
<td>11-4</td>
</tr>
<tr>
<td>Cell Arrays of Strings</td>
<td>11-7</td>
</tr>
<tr>
<td>String Comparisons</td>
<td>11-9</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching and Replacing</td>
<td>11-12</td>
</tr>
<tr>
<td>String/Numeric Conversion</td>
<td>11-13</td>
</tr>
<tr>
<td><strong>Multidimensional Arrays</strong></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Arrays</td>
<td>12-3</td>
</tr>
<tr>
<td>Computation with Multidimensional Arrays</td>
<td>12-15</td>
</tr>
<tr>
<td>Organizing Data in Multidimensional Arrays</td>
<td>12-17</td>
</tr>
<tr>
<td>Multidimensional Cell Arrays</td>
<td>12-19</td>
</tr>
<tr>
<td>Multidimensional Structure Arrays</td>
<td>12-20</td>
</tr>
<tr>
<td><strong>Structures and Cell Arrays</strong></td>
<td></td>
</tr>
<tr>
<td>Structures</td>
<td>13-3</td>
</tr>
<tr>
<td>Cell Arrays</td>
<td>13-18</td>
</tr>
<tr>
<td><strong>Classes and Objects</strong></td>
<td></td>
</tr>
<tr>
<td>Classes and Objects</td>
<td>14-2</td>
</tr>
<tr>
<td>Overloading</td>
<td>14-10</td>
</tr>
</tbody>
</table>
Object Precedence ........................................... 14-16

Inheritance ..................................................... 14-22

15

File I/O

Opening and Closing Files ................................. 15-3

Binary Files ..................................................... 15-7

Controlling Position in a File ............................ 15-9

Formatted Files .............................................. 15-12
Introduction

What Is MATLAB? .......... 1-3
The MATLAB System .......... 1-4
How to Use the Documentation Set .......... 1-6
About Simulink .......... 1-8
About Toolboxes .......... 1-8
About the Cover

The cover of this guide depicts a solution to a problem that has played a small, but interesting role in the history of numerical methods during the last 30 years. The problem involves finding the modes of vibration of a membrane supported by an L-shaped domain consisting of three unit squares. The nonconvex corner in the domain generates singularities in the solutions, thereby providing challenges for both the underlying mathematical theory and the computational algorithms. There are important applications, including wave guides, structures, and semiconductors.

Two of the founders of modern numerical analysis, George Forsythe and J.H. Wilkinson, worked on the problem in the 1950s. (See G.E. Forsythe and W.R. Wasow, Finite-Difference Methods for Partial Differential Equations, Wiley, 1960.) One of the authors of this guide (Moler) used finite differences by combinations of distinguished fundamental solutions to the underlying differential equation formed from Bessel and trigonometric functions. The idea is a generalization of the fact that the real and imaginary parts of complex analytic functions are solutions to Laplace's equation. In the early 1970s, new matrix algorithms, particularly Gene Golub's orthogonalization techniques for least squares problems, provided further algorithmic improvements.

Today, MATLAB allows us to express the entire algorithm in a few dozen lines, to compute the solution with great accuracy in a few minutes on a computer at home, and to readily manipulate color three-dimensional displays of the results. We have included our MATLAB program, membrane.m with the M-files supplied along with MATLAB.
What Is MATLAB?
MATLAB® is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for matrix laboratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called toolboxes. Very important to most users of MATLAB, toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.
The MATLAB System
The MATLAB system consists of five main parts:

The MATLAB language. This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both “programming in the small” to rapidly create quick and dirty throw-away programs, and “programming in the large” to create complete large and complex application programs. The language features are organized into six directories in the MATLAB Toolbox:

| ops | Operators and special characters. |
| lang | Programming language constructs. |
| strfun | Character strings. |
| iofun | File input/output. |
| timefun | Time and dates. |
| datatypes | Data types and structures. |

The MATLAB working environment. This is the set of tools and facilities that you work with as the MATLAB user or programmer. It includes facilities for managing the variables in your workspace and importing and exporting data. It also includes tools for developing, managing, debugging, and profiling M-files, MATLAB’s applications. The working environment features are located in a single directory.

| general | General purpose commands. |

Handle Graphics®. This is the MATLAB graphics system. It includes high-level commands for 2-D and 3-D data visualization, image processing, animation, and presentation graphics. It also includes low-level commands that allow you to fully customize the appearance of graphics as well as to build complete
graphical user interfaces (GUIs) for your MATLAB applications. The graphics functions are organized into five directories in the MATLAB Toolbox.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph2d</td>
<td>Two-dimensional graphs.</td>
</tr>
<tr>
<td>graph3d</td>
<td>Three-dimensional graphs.</td>
</tr>
<tr>
<td>specgraph</td>
<td>Specialized graphs.</td>
</tr>
<tr>
<td>graphi cs</td>
<td>Handle Graphics.</td>
</tr>
<tr>
<td>u t ool s</td>
<td>Graphical user interface tools.</td>
</tr>
</tbody>
</table>

**The MATLAB mathematical function library.** This is a vast collection of computational algorithms ranging from elementary functions like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms. The math and analytic functions are organized into eight directories in the MATLAB Toolbox.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>el nat</td>
<td>Elementary matrices and matrix manipulation.</td>
</tr>
<tr>
<td>el fun</td>
<td>Elementary math functions.</td>
</tr>
<tr>
<td>specfun</td>
<td>Specialized math functions.</td>
</tr>
<tr>
<td>matfun</td>
<td>Matrix functions – numerical linear algebra.</td>
</tr>
<tr>
<td>datafun</td>
<td>Data analysis and Fourier transforms.</td>
</tr>
<tr>
<td>pol yfun</td>
<td>Interpolation and polynomials.</td>
</tr>
<tr>
<td>funfun</td>
<td>Function functions and ODE solvers.</td>
</tr>
<tr>
<td>sparfun</td>
<td>Sparse matrices.</td>
</tr>
</tbody>
</table>

**The MATLAB Application Program Interface (API).** This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.
How to Use the Documentation Set

MATLAB comes with an extensive set of documentation consisting of an online Help facility and online Function Reference as well as printed manuals. The full set of printed documentation includes the following titles:

- The MATLAB Installation Guide describes how to install MATLAB on your platform.
- Getting Started with MATLAB explains how to get started with the fundamentals of MATLAB.
- Using MATLAB provides in depth material on the MATLAB language, working environment, and mathematical topics.
- Using MATLAB Graphics describes how to use MATLAB’s graphics and visualization tools.
- The MATLAB Application Program Interface Guide explains how to write C or Fortran programs that interact with MATLAB.
- The MATLAB 5.1 New Features Guide provides information useful in making the transition from MATLAB 4.x to 5.1.

<table>
<thead>
<tr>
<th>What I Want</th>
<th>What I Should Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>I need to install MATLAB.</td>
<td>See the Installation Guide for your platform.</td>
</tr>
<tr>
<td>I’m new to MATLAB and want to learn it fast.</td>
<td>Start by reading Getting Started with MATLAB. The most important things to learn are how to enter matrices, how to use the : (colon) operator, and how to invoke functions. After you master the basics, you can access the rest of the documentation as needed, or you can use online help and the demonstrations to learn other commands.</td>
</tr>
<tr>
<td>I’m upgrading from MATLAB 4.</td>
<td>Read the MATLAB 5 New Features Guide to find out about the new features in MATLAB 5. Pay special attention to the “Upgrading to MATLAB 5” section for how to convert your M-files. You should then refer to Using MATLAB and Using MATLAB Graphics for specific details about the new features.</td>
</tr>
<tr>
<td>What I Want</td>
<td>What I Should Do</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>I want to know how to use a specific function.</td>
<td>Use the online Help facility. You can use the M-file help window to get brief online help or access the full function reference via the Web-based Help Desk. These are available using the commands helpwin and helpdesk or from the Help menu on the PC and Macintosh. The function reference is also available on the Help Desk in PDF format if you want to print out any of the function descriptions in high-quality form.</td>
</tr>
<tr>
<td>I want to find a function for a specific purpose but I don't know its name.</td>
<td>There are three choices</td>
</tr>
<tr>
<td></td>
<td>1 Use lookfor (e.g. lookfor inverse) from the command line.</td>
</tr>
<tr>
<td></td>
<td>2 Use the online keyword search from the Help Desk.</td>
</tr>
<tr>
<td></td>
<td>3 Visit The MathWorks Web site and see if there is a user-contributed file to solve your problem.</td>
</tr>
<tr>
<td>I want to learn about a specific topic like sparse matrices, ordinary differential equations, or cell arrays.</td>
<td>See the appropriate chapter in Using MATLAB.</td>
</tr>
<tr>
<td>I want to know what functions are available in a general area.</td>
<td>Use the help window (type helpwin or select from Help menu) to see a table of contents with functions grouped by subject area, or use the Help Desk (type helpdesk or select from Help menu) to see the Function Reference grouped by subject.</td>
</tr>
<tr>
<td>I have a problem I want help with.</td>
<td>For tips and troubleshooting problems, use the Help Desk (type helpdesk or select from Help menu) to visit the Technical Support section of The MathWorks Web site (<a href="http://www.mathworks.com">www.mathworks.com</a>) and use the Solution Search Engine to search the Technical Support database of problem solutions.</td>
</tr>
</tbody>
</table>
Introduction

Simulink®, a companion program to MATLAB, is an interactive system for simulating nonlinear dynamic systems. It is a graphical mouse-driven program that allows you to model a system by drawing a block diagram on the screen and manipulating it dynamically. It can work with linear, nonlinear, continuous-time, discrete-time, multivariable, and multirate systems.

Blocksets are add-ins to Simulink that provide additional libraries of blocks for specialized applications like communications, signal processing, and power systems.

Real-time Workshop® is a program that allows you to generate C code from your block diagrams and to run it on a variety of real-time systems.

About Toolboxes
MATLAB features a family of application-specific solutions called toolboxes. Very important to most users of MATLAB, toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment in order to solve particular classes of problems. Many toolboxes are available from The MathWorks. Some of these are listed on the following page; contact The MathWorks or visit www.mathworks.com for a complete up-to-date list.

<table>
<thead>
<tr>
<th>What I Want</th>
<th>What I Should Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>I want to report a bug or make a suggestion.</td>
<td>Use the Help Desk (type helpdesk or select from Help menu) or send e-mail to <a href="mailto:bugs@mathworks.com">bugs@mathworks.com</a> or <a href="mailto:suggest@mathworks.com">suggest@mathworks.com</a></td>
</tr>
<tr>
<td>I want to contact The MathWorks Technical Support.</td>
<td>Use the Help Desk (type helpdesk or select from Help menu) to submit an e-mail help request form describing your question or problem.</td>
</tr>
</tbody>
</table>
The MATLAB Product Family

How The MathWorks products fit together

MATLAB is the foundation for all The MathWorks products. MATLAB combines numeric computation, 2-D and 3-D graphics, and language capabilities in a single, easy-to-use environment.

MATLAB Extensions are optional tools that support the implementation of systems developed in MATLAB.

Toolboxes are libraries of MATLAB functions that customize MATLAB for solving particular classes of problems. Toolboxes are open and extensible; you can view algorithms and add your own.

Simulink is a system for nonlinear simulation that combines a block diagram interface and “live” simulation capabilities with the core numeric, graphics, and language functionality of MATLAB.

Simulink Extensions are optional tools that support the implementation of systems developed in Simulink.

Blocksets are collections of Simulink blocks designed for use in specific application areas.

Contact The MathWorks or visit www.mathworks.com for a complete up-to-date list.
Using the Environment

The Command Window .............................. 2-3
Editing M-Files .................................... 2-8
Preference Setting ................................. 2-13
Command History ................................. 2-17
The MATLAB Workspace ....................... 2-18
The MATLAB Search Path ....................... 2-26
Help and Online Documentation .............. 2-31
Disk File Manipulation and Shell Escape .... 2-35
Data Import/Export ............................... 2-36
The Startup M-File ............................... 2-41
Memory Utilization ............................... 2-42
License Manager Administration (UNIX) ...... 2-44
MATLAB is both a language and a working environment. This chapter focuses on the MATLAB working environment. As a working environment, MATLAB includes facilities for managing the variables in your workspace and for importing and exporting data. MATLAB also includes tools for developing and managing M-files, MATLAB's applications.

**Starting MATLAB**

To run MATLAB on a PC or Macintosh, double-click on the MATLAB icon. To run MATLAB on a UNIX system, type `matlab` at the operating system prompt.

To quit MATLAB at any time, type `quit` at the MATLAB prompt. On the PC and Macintosh, you may prefer using **Exit** or **Quit** from the **File** menu.

**Shortcuts and Aliases**

On the PC, the installer creates a shortcut to the program file in the installation directory. You can move this shortcut to your desktop if you want. Double-click on this shortcut icon to start MATLAB.

On the Macintosh, the best way to run MATLAB is to create an alias to the program file:

1. Locate the MATLAB application.
2. Single-click on the file to select it, and choose **Make Alias** from the **File** menu.
3. Drag this new alias file to your desktop.

You can now double-click on this alias file to start MATLAB.

On UNIX, you may find it useful to create a new directory off of your home directory called **matlab**. For example, if your username is **fred**, you might type:

```
mkdir /home/fred/matlab
```

MATLAB knows to search in this location for M-files you create.
The Command Window

The Command Window is the main window in which you communicate with the MATLAB interpreter. On UNIX systems, the Command Window is the terminal window from which you start MATLAB. On the PC and Macintosh, MATLAB provides a special window with platform-dependent features.

The MATLAB interpreter displays a prompt (>>) indicating that it is ready to accept commands from you. For example, to enter a 3-by-3 matrix, you can type

\[ A = [1 \ 2 \ 3; \ 4 \ 5 \ 6; \ 7 \ 8 \ 10] \]

When you press the Enter or Return key, MATLAB responds with

\[ A = \]

\[ 1 \ \ 2 \ \ 3 \]
\[ 4 \ \ 5 \ \ 6 \]
\[ 7 \ \ 8 \ \ 10 \]

To invert this matrix, enter

\[ B = \text{inv}(A) \]

MATLAB responds with the result.

Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse commands you have typed earlier. For example, suppose you mistakenly enter

\[ \rho = (1 + \sqrt{5})/2 \]

You have misspelled sqrt. MATLAB responds with

Undefined function or variable 'sqt'.

Instead of retyping the entire line, simply press the \[ \uparrow \] key. The misspelled command is redisplayed. Use the \[ \leftarrow \] key to move the cursor over and insert the missing \[ r \]. Repeated use of the \[ \uparrow \] key recalls earlier lines.

The commands you enter during a MATLAB session are stored in a buffer. You can use smart recall to recall a previous command whose first few characters
you specify. For example, typing the letters `plot` and pressing the ↑ key recalls
the last command that started with `plot`, as in the most recent plot command.

The complete list of arrow and control keys provides additional control. Many
of these keys should be familiar to users of the EMACS editor.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td><code>ctrl-p</code> Recall previous line.</td>
</tr>
<tr>
<td>↓</td>
<td><code>ctrl-n</code> Recall next line.</td>
</tr>
<tr>
<td>←</td>
<td><code>ctrl-b</code> Move back one character.</td>
</tr>
<tr>
<td>→</td>
<td><code>ctrl-f</code> Move forward one character.</td>
</tr>
<tr>
<td><code>ctrl-</code> →</td>
<td><code>ctrl-r</code> Move right one word.</td>
</tr>
<tr>
<td><code>ctrl-</code> ←</td>
<td><code>ctrl-l</code> Move left one word.</td>
</tr>
<tr>
<td><code>option-</code> →</td>
<td>Move right one word.</td>
</tr>
<tr>
<td><code>option-</code> ←</td>
<td>Move left one word.</td>
</tr>
<tr>
<td><code>home</code></td>
<td><code>ctrl-a</code> Move to beginning of line.</td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>ctrl-e</code> Move to end of line.</td>
</tr>
<tr>
<td><code>esc</code></td>
<td><code>ctrl-u</code> Clear line.</td>
</tr>
<tr>
<td><code>del</code></td>
<td><code>ctrl-d</code> Delete character at cursor.</td>
</tr>
<tr>
<td><code>backspace</code></td>
<td><code>ctrl-h</code> Delete character before cursor.</td>
</tr>
<tr>
<td><code>ctrl-k</code></td>
<td>Delete (kill) to end of line.</td>
</tr>
</tbody>
</table>

### Interrupting a Running Program

You can interrupt a running program by pressing **Ctrl-c** at any time. On UNIX systems, program execution will terminate immediately. On other platforms you may have to wait until an executing built-in function or MEX-file has finished its operation.
The format Command

The format command controls the numeric format of the values displayed on the screen. The command only affects how numbers are displayed, not how MATLAB computes or saves them. Here are the different formats, together with the output produced from a two-element vector with components of different magnitudes.

```matlab
x = [4/3 1.2345e-6]

format short
1.3333    0.0000

format short e
1.3333e+000  1.2345e-006

format short g
1.3333 1.2345e-006

format long
1.33333333333333   0.00000123450000

format long e
1.333333333333333e+000    1.234500000000000e-006

format long g
1.33333333333333               1.2345e-006

format bank
1.33          0.00
```
format rat
4/3
1/810045
format hex
3ff5555555555555
3eb4b6231abfd271
If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats.

In addition to the `format` commands shown above

format compact
suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window. If you want more control over the output format, use the `sprintf` and `fprintf` functions.

**Suppressing Output**

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example

A = magic(100);

**Long Command Lines**

If a statement does not fit on one line, use three periods, . . ., followed by Return or Enter to indicate that the statement continues on the next line. For example

s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 . . .
- 1/8 + 1/9 - 1/10 + 1/11 - 1/12;

Blank spaces around the =, +, and - signs are optional, but they improve readability. The maximum number of characters allowed on a single line is 4096.
Command Window Toolbar

On the PC and Macintosh, a Command Window toolbar provides easy access to common operations:
Editing M-Files

The PC Editor/Debugger and the Macintosh M-file Editor provide basic text editing operations as well as access to M-file debugging tools.

One way to edit an M-file is from the MATLAB command line using the `edit` command. For example

```
edit poof
```

opens an editor for the file `poof.m`. On the PC and Macintosh, this opens the built-in editor unless you’ve selected a different editor in the Preferences dialog (accessible from the File menu). On UNIX workstations, `edit` starts the editor specified by the environment variable EDIT.

On the PC and Macintosh, a second way to open an M-file for editing is from the Command Window menu:

- To open a new M-file for editing, select New from the File menu or click on the New File icon on the toolbar.
- To open an existing M-file for editing, select Open from the File menu or click on the Open File icon on the toolbar.

The editors for both the PC and Macintosh platforms provide syntax highlighting capability. As you type into the editor, the text is colored according to the type of text entered. On the Macintosh you can choose to override the default color settings. The various types of text are:

- Comments
- Keywords
- Incomplete strings
- Complete strings
- Other text
PC Editor/ Debugger

Toolbar
The toolbar for the PC Editor/Debugger looks like:

Menus
When you are editing files, the editor allows you to go directly to a specific line. Select the Go To Line item on the Edit menu:
Two interesting operations on the View menu are **Evaluate Selection**, which evaluates an expression and places the answer in the Command Window:

and **Auto Indent Selection**, which indents the selected text according to MATLAB syntax:
Choosing **Options** from the **View** menu provides access to a dialog box that allows you to control text formatting within the editor:

![Options dialog box](image)

**Macintosh M-File Editor**

The toolbar for the Macintosh M-file Editor looks like

![Macintosh M-File Editor toolbar](image)
Some useful features of the Macintosh editor include:

- **Command**-clicking in the title of an editor window displays a pop-up menu containing the full path to the M-file. Selecting a folder from the pop-up menu opens that folder in the Finder.

- Selecting text in an editor window and pressing the **Enter** key evaluates that text in the Command Window.

- When the **Balance Typing** option in **Editor Preferences** (located under the **File** menu on the main toolbar) is checked, parentheses ( ), brackets [ ], and braces { } are balanced; for example, typing a close parenthesis briefly highlights the matching open parenthesis.

**Editor Linking for Error Displays**

On the Macintosh, you can automatically open the editor to the point where an error occurred by placing the cursor on an error message and pressing the **Enter** key (not the **Return** key). For example, given the error message

```
??? Undefined function or variable 'c'.
Error in ==> HD:Desktop Folder:myfile.m
On line 3  ==> c & 3;
```

place the cursor on the Error in line and press the **Enter** key to open myfile.m with line 3 selected.
Preference Setting

Setting PC Preferences

Preference setting allows you to control aspects of the appearance and operation of the environment tools. On the PC you access the Preferences dialog boxes by choosing Preferences from the File menu:
Setting Macintosh Preferences

On the Macintosh choose Preferences from the main toolbar to access the Preferences dialog boxes. The Preferences dialog has groups of options that are accessed via a “tab” model:

**General Preferences**

Setting General preferences enables you to:

- Speed up launch and initial runtime by saving path cache information to disk.
- Yield CPU time to other applications during M-code execution.
- Echo commands in M-files.
- Enable live matching of parenthesis, brackets, and braces.
- Enable live scrolling when clicking and dragging the scrollbar thumb.
- Revert to command line debugger.
- Set numerical output format.
- Set numerical output display spacing.
Command Window Preferences
Setting Command Window preferences enables you to:

• Choose the font to use in the Command Window.
• Choose the font size to use in the Command Window.
• Set the number of spaces that a tab represents in the Command Window.
• Set the font style as bold in the Command Window.
• Set the font style as italic in the Command Window.
• Turn on the Command Window toolbar.

Editor Preferences
Setting Editor preferences enables you to:

• Set the font style as bold in new M-file Editor documents.
• Set the font style as italic in new M-file Editor documents.
• Turn on automatic indenting in the M-file Editor.
• Turn on smart indenting in the M-file Editor.
• Turn on the line number display in the M-file Editor.
• Turn on the M-File Editor window toolbar.
• Substitute spaces for tabs in the M-file Editor.
• Display the current M-file Editor.
• Select an editor of your choice to use in place of the built-in Editor.
• Use the built-in M-file Editor.

Help Preferences
Setting Help preferences enables you to:

• Display the Web browser that you want to use for help.
• Change your preferred Web browser.
• Display the path to your help folder.
• Change the path to your help folder.
Printing Preferences
Setting **Printing** preferences enables you to:
- Select the format to use when printing MATLAB figure windows.
- Select the rendering method to use when printing MATLAB figure windows.
- Force figure window background to white when printing.

Saving Preferences
Setting **Saving** preferences enables you to:
- Select the file format to use when saving figure windows.
- Select the preview type to use when saving figure windows.
- Force figure window background to white when saving.

Copying Preferences
Setting **Copying** preferences enables you to:
- Set figure window capture type to be bitmap or Macintosh PICT.
- Force figure window background to white when copying.
- Enable honoring figure size properties.
- Select embedded PostScript style when copying.

Syntax Coloring
Setting **Syntax Coloring** preferences enables you to:
- Choose the main text color.
- Choose the comment text color.
- Choose the keyword text color.
- Choose the string text color.
- Choose the incomplete string text color.
- Reset all text colors to their factory defaults.
Command History

Pressing the **Command History** button ( ) on the Macintosh Command Window toolbar displays the **Command History** window:

The **Command History** window contains a list of all previous commands that have been executed. The list is limited to approximately 30,000 bytes of text. The oldest commands are removed when the list approaches this limit.
The MATLAB Workspace

The MATLAB workspace contains a set of variables (named arrays) that you can manipulate from the MATLAB command line. You can use the `who` and `whos` commands to see what is currently in the workspace. The `who` command gives a short list, while the `whos` command also gives size and data type information.

Here is the output produced by `whos` on a workspace containing eight variables of a variety of different data types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4x4</td>
<td>128</td>
<td>double array</td>
</tr>
<tr>
<td>D</td>
<td>3x5</td>
<td>120</td>
<td>double array</td>
</tr>
<tr>
<td>M</td>
<td>10x1</td>
<td>40</td>
<td>cell array</td>
</tr>
<tr>
<td>S</td>
<td>1x3</td>
<td>628</td>
<td>struct array</td>
</tr>
<tr>
<td>h</td>
<td>1x11</td>
<td>22</td>
<td>char array</td>
</tr>
<tr>
<td>n</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>s</td>
<td>1x5</td>
<td>10</td>
<td>char array</td>
</tr>
<tr>
<td>v</td>
<td>1x14</td>
<td>28</td>
<td>char array</td>
</tr>
</tbody>
</table>

Grand total is 93 elements using 984 bytes

To delete all existing variables from the workspace, enter

```
clear
```

Workspace Browsers

On the PC and Macintosh, a Workspace Browser lets you view the contents of the current MATLAB workspace. It provides a graphical representation of the `whos` display. To open the Workspace Browser, select Show Workspace from the File menu or click on the Workspace Browser button on the toolbar.
The Workspace Browser on the PC

- To clear variables, select the variables and click on Delete.
- To close the window, click Close.

You can resize the columns of information by dragging the column header borders, just like in the Windows 95 Explorer. The workspace is sorted by variable name. Sorting by other fields is not currently supported.

To rename a variable, first select it, then click on its name. (Note that double-clicking on the name does nothing.) After a short delay, you can type a new name; press Enter to complete the name change.
The Workspace Browser on the Macintosh

By default, the workspace is sorted by variable name. Click on the appropriate label to sort the workspace by size, bytes, or class. Option-click on a label to reverse-sort by that label.

Editing Arrays on the Macintosh

You invoke the Macintosh Array Editor by double-clicking an icon in the Workspace Browser. With the Array Editor you can view and edit two-dimensional real and complex double arrays, row vector character arrays, and row or column vector cell arrays of strings. If the Array Editor does not support a variable type, double-clicking on the variable in the Workspace Browser displays the variable in the Command Window. This tool can be especially useful when debugging M-functions using the M-file Debugger.
To edit the contents of a cell, select the cell and begin typing. When you edit a double array, you can enter values or expressions. If you enter an expression, it is evaluated and the result placed in the cell:

```
   sin(pi)
```

Any text typed when you edit character arrays or cell arrays of strings is placed in the string.

The Array Editor supports Cut, Copy, and Paste operations on selections in the window:

![Array Editor toolbar](image)

The data is placed on the clipboard as tab-delimited text, so that it can be pasted into other applications. Undo of in-cell edits is supported. Undo of Cut and Paste operations is not supported in this release.

The **Format** pop-up menu allows you to change the format of the output display as if you were using the `format` command:

```
   %f
   %e
   %g
```

Supported formats include short, short e, short g, long, long e, and long g.

You can drag column and row titles within the window to change the order of columns and rows in the array:

![Array Editor column and row titles](image)

In addition to the name of the variable, the window title contains information about its scope, including the name of the workspace containing the variable, and, for variables inside M-functions, the level of the stack:

```
x (myfact), Stack Level: 4
```
Loading and Saving the Workspace
MATLAB’s `save` and `load` commands let you save the contents of the MATLAB workspace at any time during a session, and then reload the data back into MATLAB during that session or a later one. `load` and `save` can also import and export ASCII data files.

Saving the Workspace
The `save` command saves the contents of the workspace into a binary MAT-file that you can read back later with the `load` command. For example

```
save june10
```

saves the entire workspace contents in the file `june10.mat`. If desired, you can save only certain variables by specifying the variable names after the filename. For example,

```
save june10 x y z
```

saves only variables `x`, `y`, and `z`.

On the PC and Macintosh, the `save` operation is also available by selecting `Save Workspace As` from the `File` menu.

**NOTE** The MATLAB Application Program Interface Guide provides details on reading and writing MAT-files from external C or Fortran programs.
Specifying File Format
You can control the format in which save stores data by appending flags to the filename/variable name list:

- **-mat** Use binary MAT-file form (default).
- **-ascii** Use 8-digit ASCII form.
- **-ascii -double** Use 16-digit ASCII form.
- **-ascii -double -tabs** Delimit array elements with tabs.
- **-v4** Save in MATLAB version 4 format.
- **-append** Append data to existing MAT-file.

If you use the v4 flag, you can only save data constructs that are compatible with V4 versions of MATLAB. That is, you cannot save structures, cell arrays, multidimensional arrays, or objects.

When you save workspace contents in ASCII format, save only one variable at a time. If you save more than one variable, MATLAB will create the ASCII file, but you will be unable to load it back into MATLAB later using load.

Loading the Workspace
The load command loads a MAT-file that you have previously created with save. For example

```
load june10
```

loads june10.mat into the workspace. If the saved MAT-file june10 contains the variables A, B, and C, then loading june10 places the variables A, B, and C back into the workspace. If the variables already existed in the workspace, they are overwritten.

If your MAT-file has a filename extension other than .mat, you must use the -mat switch or else MATLAB expects the file to be ASCII text format.

```
load filename -mat
```

On the PC and Macintosh, the load operation is also available by selecting Load Workspace from the File menu.
Loading ASCII Data Files
The `load` command also imports ASCII data files. It reads the contents of the file into a variable with the same name as the file (without the extension). For example

```
load tides.dat
```

creates a variable named `tides` in the workspace. If the ASCII data file has `m` lines with `n` values on each line, the result is an `m`-by-

`n` numeric array.

Filenames Stored in String Variables
If the file and variable names you are working with are stored in string variables, you can use command/function duality to call `load` and `save` as functions. In this case, the input arguments appear in the same order as they would at the command line. For example, the statements

```
save('myfile','VAR1','VAR2')
A = 'myfile';
load(A)
```

are the same as

```
save myfile VAR1 VAR2
load myfile
```

To load or save multiple files with the same prefix and successive integer suffixes, use a loop. For example, this code saves the squares of the numbers 1 through 10 in files `data1` through `data10`:

```
file = 'data';
for i = 1:10
    j = i.^2;
    save([file int2str(i)],'j');
end
```

Wildcards
The `load` and `save` commands let you specify a wildcard character (*) to search for patterns of variable names. For example

```
save rundate x*
```
saves all variables in the workspace that start with x in the file rundata.mat.
Similarly

```matlab
load testdata ex1*95
```

loads from testdata.mat all the variables whose first three characters are 'ex1' and last two characters are '95', regardless of the characters in between them.
The MATLAB Search Path

MATLAB has a search path that it uses to find M-files. MATLAB’s M-files are organized in directories or folders on your file system. Many of these directories of M-files are provided along with MATLAB, while others are available separately as Toolboxes.

If you enter the name \texttt{foo} at the MATLAB prompt, the MATLAB interpreter:

1. Looks for \texttt{foo} as a variable.
2. Checks for \texttt{foo} as a built-in function.
3. Looks in the current directory for a file named \texttt{foo.m}
4. Searches the directories on the search path for \texttt{foo.m}

While the actual search rules are more complicated because of the restricted scope of private functions, subfunctions, and object-oriented functions, this simplified perspective is accurate for the ordinary M-files that you usually work with.

If you have more than one function with the same name, only the first one in search path order is found; other functions with the same name are considered to be shadowed and cannot be executed.

Changing the Search Path

You can display and change the search path for the duration of your current session using the \texttt{path}, \texttt{addpath}, and \texttt{rmpath} functions.

- \texttt{path}, by itself, returns the current search path.
- \texttt{path(s)}, where \texttt{s} is a string, sets the path to \texttt{s}.
- \texttt{addpath /home/lib} and \texttt{path(path, '/home/lib')} both append a new directory to the path.
- \texttt{rmpath /home/lib} removes the path /home/lib.

On UNIX and the PC the default search path remembered between sessions is defined in the file \texttt{pathdef.m} in the directory named \texttt{local} on your system. \texttt{pathdef} executes automatically each time you start MATLAB. On UNIX workstations you may not have file system permission to edit \texttt{pathdef.m}.

this case, put `path` and `addpath` commands in your `startup.m` file to change your path defaults. On the PC you can directly edit `pathdef.m` with your text editor. The PC also provides a Path Browser with a convenient interface for viewing and changing the search path.

The Macintosh stores preference information in the `Preferences` folder under the `System` folder.

**The Current Directory**

MATLAB maintains a current directory for the purpose of working with M-files and MAT-files.

On the PC, the initial current directory is specified in the shortcut file you use to start MATLAB. Right-click on the shortcut file, and select Properties to change the default.

On the Macintosh, the initial current directory is the folder in which MATLAB is installed.

On UNIX systems, the initial current directory is the directory you are in on your UNIX file system when you invoke MATLAB.

To display your current directory, use the `cd` command with no arguments. For example on UNIX:

```
cd
/home/roger
```

To change your current directory, use `cd` with a path. For example, on a PC

```
cd \bigproj\phase1
```

On the PC and Macintosh you can also change the current directory from the Path Browser.

**Viewing Files on the Search Path**

You’ve already seen how `path` displays the search path. The `what` command drills down into a specific directory on the path to tell you what MATLAB files are there. With no arguments, `what` displays the files in the current directory.

`what`
With a full or partial path, what lists the files in any directory on the path
what matlab/elfun
To see the code in a specific M-file, use type
type rank
To edit the M-file, use editedit rank

The Path Browser
On the PC and Macintosh, a Path Browser lets you view and modify MATLAB’s search path and see all of its files. To open the Path Browser, select Set Path from the File menu or click on the Path Browser button on the toolbar.

The Path Browser on the PC

To move a directory to a different position on the path, drag it to the desired location.
NOTE If you change the path from the Command Window, the Path Browser’s contents will be out of date until you use the Refresh button.

All changes take effect in MATLAB for the duration of the current session. Click on Save Settings to save the new path permanently.

The Path Browser on the Macintosh

To add files and folders to the path list, drag them from the directory listing to the path list. If you have checked the Add subfolders box, all subfolders of the selected directory will be added to the path as well.

In addition to the using Remove button, you can remove items from the path by dragging them to the trash.
Change the current working directory by dragging an item from the path list or from the directory list into the current MATLAB directory display. Change the directory listing by double-clicking the current MATLAB directory display, or by double-clicking an item in the path list.
Help and Online Documentation

There are several different ways to access online information about MATLAB functions.

- The help command
- The help window
- The lookfor command
- The MATLAB Help Desk
- Printing online reference pages
- The MathWorks Web Site

The help Command

The help command is the most basic way to determine the syntax and behavior of a particular function. Information is displayed directly in the command window. For example

```matlab
help magic
```

displays

```
MAGIC Magic square.
MAGIC(N) is an N-by-N matrix constructed from
the integers 1 through N^2 with equal row,
column, and diagonal sums.
Produces valid magic squares for N = 1, 3, 4, 5, . . .
```

NOTE: MATLAB online help entries use uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, always use the corresponding lowercase characters since MATLAB is case sensitive and all function names are actually in lowercase.

All the MATLAB functions are organized into logical groups, and MATLAB’s directory structure is based on this grouping. For instance, all the linear
algebra functions reside in the matfun directory. To list the names of all the functions in that directory, with a brief description of each:

```
help matfun
```

Matrix functions - numerical linear algebra.

Matrix analysis.

```
norm - Matrix or vector norm
normest - Estimate the matrix 2-norm
...
```

The command

```
help
```

by itself, lists all the directories, with a description of the function category each represents:

```
matlab/general
matlab/ops
...
```

The Help Window

The MATLAB help window is available on PCs and Macintoshes by selecting the Help Window option under the Help menu, or by clicking the question mark on the menu bar. It is also available on all computers by typing

```
helpwin
```

To use the help window on a particular topic, type

```
help win topic
```

The help window gives you access to the same information as the help command, but the window interface provides convenient links to other topics.

The lookfor Command

The lookfor command allows you to search for functions based on a keyword. It searches through the first line of help text, which is known as the H1 line, for each MATLAB function, and returns the H1 lines containing a specified
keyword. For example, MATLAB does not have a function named \texttt{inverse}. So the response from

\begin{verbatim}
help inverse
\end{verbatim}

is

\begin{verbatim}
inverse.m not found.
\end{verbatim}

But

\begin{verbatim}
lookfor inverse
\end{verbatim}

finds over a dozen matches. Depending on which toolboxes you have installed, you will find entries like

\begin{verbatim}
INVHILB Inverse Hilbert matrix.
ACOSH Inverse hyperbolic cosine.
ERFINV Inverse of the error function.
INV Matrix inverse.
PINV Pseudoinverse.
IFFT Inverse discrete Fourier transform.
IFFT2 Two-dimensional inverse discrete Fourier transform.
ICCEPS Inverse complex cepstrum.
IDCT Inverse discrete cosine transform.
\end{verbatim}

Adding \texttt{--all} to the \texttt{lookfor} command searches the entire help entry, not just the \texttt{H1} line.

\section*{The Help Desk}

The MATLAB Help Desk provides access to a wide range of help and reference information stored on a disk or CD-ROM in your local system. Many of the underlying documents use HyperText Markup Language (HTML) and are accessed with an Internet Web browser such as Netscape Navigator or Microsoft Internet Explorer. The Help Desk process can be started on PCs and Macintoshes by selecting the \textbf{Help Desk} option under the \textbf{Help} menu, or, on all computers, by typing

\begin{verbatim}
hel pdesk
\end{verbatim}

All of MATLAB's operators and functions have online reference pages in HTML format, which you can reach from the Help Desk. These pages provide more details and examples than the basic \texttt{help} entries. HTML versions of other
documents, including this manual, are also available. A search engine, running on your own machine, can query all the online reference material.

**The doc Command**

If you know the name of a specific function, you can view its reference page directly. For example, to get the reference page for the `eval` function, type

```
  doc eval
```

The `doc` command starts your Web browser, if it is not already running.

**Printing Online Reference Pages**

Versions of the online reference pages are also available in Portable Document Format (PDF) through the Help Desk. These pages are processed by Adobe's Acrobat reader. They reproduce the look and feel of the printed page, complete with fonts, graphics, formatting, and images. This is the best way to get printed copies of reference material.

**The MathWorks Web Site**

If your computer is connected to the Internet, the Help Desk provides connections to The MathWorks, the home of MATLAB. You can use electronic mail to ask questions, make suggestions, and report possible bugs. You can also use the Solution Search Engine at The MathWorks Web site to query an up-to-date data base of technical support information.

Alternatively you can point your Web browser directly at [www.mathworks.com](http://www.mathworks.com) to access The MathWork Web site.
Disk File Manipulation and Shell Escape

The commands `dir`, `type`, `delete`, and `cd` implement a set of generic operating system commands for manipulating files. This table indicates how these commands map to other operating systems:

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>MS-DOS</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dir</code></td>
<td><code>dir</code></td>
<td><code>ls</code></td>
</tr>
<tr>
<td><code>type</code></td>
<td><code>type</code></td>
<td><code>cat</code></td>
</tr>
<tr>
<td><code>delete</code></td>
<td><code>delete</code></td>
<td><code>rm</code></td>
</tr>
<tr>
<td><code>cd</code></td>
<td><code>chdir</code></td>
<td><code>cd</code></td>
</tr>
</tbody>
</table>

For most of these commands, you can use pathnames, wildcards, and drive designators in the usual way.

Running External Programs

The exclamation point character `!` is a shell escape and indicates that the rest of the input line is a command to the operating system (or to the Finder on the Macintosh). This is quite useful for invoking utilities or running other programs without quitting from MATLAB. On UNIX, for example

```
! vi darwin.m
```

invokes the `vi` editor for a file named `darwin.m`. After you quit the program, the operating system returns control to MATLAB.

See the commands `unix` and `dos` in online help to run external programs that return results and status.

Under UNIX, pressing `Ctrl-Z` suspends the current process and brings up a new shell from which an editor can be invoked. When editing is finished, the command `%matlab` returns MATLAB to the foreground.
Data Import/ Export

There are many ways to move data between MATLAB and other applications. In most cases, you can simply use MATLAB's native data exchange capabilities to read in or write out files. For more complicated data sets, you may want to create your own C or Fortran program to read or write a file.

Importing Data into MATLAB

You can introduce data from other programs into MATLAB using several methods. The best method for importing data depends on the amount and format of the data.

• **Enter data as an explicit list of elements.** If you have a small amount of data, it is easy to type the data explicitly using brackets ([ ]). This method is awkward for larger amounts of data because you can't edit your input if you make a mistake, but must correct it using assignment statements. See the Getting Started with MATLAB guide for more information on this technique.

• **Create data in an M-file.** Use a text editor to create an M-file that enters the data as an explicit list of elements. This method is useful when the data is not already in digital form and must be entered anyway. Essentially the same as the first method, this method has the advantage of allowing you to use your editor to change the data and to fix mistakes. You can then just rerun the M-file to re-enter the data.

• **Load data from an ASCII data file.** An ASCII data file stores the data in ASCII form, with each row having the same number of values and terminating with new lines (carriage returns), and with spaces separating the numbers. You can edit ASCII data files using a normal text editor. You can read ASCII data files directly into MATLAB using the `load` function. This creates a variable whose name is the same as the filename. See "section Loading and Saving the Workspace" on page 2-22 for details on `load`. You can also use `dlmread` if you need to specify alternate value delimiters. `dlmread` is discussed on page 2-38.

• **Read data using `fopen`, `fread`, and MATLAB's file I/O functions.** This method is useful for loading data files from other applications that have their own established file formats. These functions are discussed in detail in Chapter 15.
• Use a specialized file reader function such as wk1read, dlmread, wavread, or imread for application-specific formats.
  
  - dlmread Read ASCII data file.
  - wk1read Read spreadsheet (WK1) file.
  - imread Read image from graphics file.
  - auread Read SUN (".au") sound file.
  - wavread Read Microsoft WAVE (".wav") sound file.
  - readsnd Read SND resources and files (Macintosh only).

• Develop a MEX-file to read the data. This is the best method if C or Fortran routines are already available for reading data files from other applications. See the MATLAB Application Program Interface Guide for more information.

• Develop a Fortran or C program to translate your data into MAT-file format and then read the MAT-file into MATLAB with the load command. See the MATLAB Application Program Interface Guide for more information.

Exporting Data From MATLAB

There are several methods for getting MATLAB data to other applications:

• For small arrays, use the diary command to create a diary file and display the variables, echoing them into this file. The output of diary includes the MATLAB commands used during the session, which is useful for inclusion in documents and reports. You can use your text editor to edit the diary file, removing unwanted text.

• Save the data in ASCII form using the save command with the -ascii option. See “section Loading and Saving the Workspace” on page 2-22 for details on save. You can also use dlmwrite if you need to specify alternate value delimiters. dlmwrite is discussed on page 2-38.

• Write the data in a special format using fopen, fwrite, and the other low-level I/O functions. This method is useful for writing data files in the file
formats required by other applications. These functions are discussed in detail in Chapter 15.

- **Use a specialized file write function** such as `wk1write`, `dlmwrite`, `wavwrite`, or `imwrite` for application-specific formats.

  - `dlmwrite` Write ASCII data file.
  - `wk1write` Write spreadsheet (WK1) file.
  - `imwrite` Write image to graphics file.
  - `auwrite` Write SUN (".au") sound file.
  - `wavwrite` Write Microsoft WAVE (".wav") sound file.
  - `writesnd` Write SND resources and files (Macintosh only).

- **Develop a MEX-file to write the data.** This is the best method if C or Fortran routines are already available for writing data files in the form needed by other applications. See the MATLAB Application Program Interface Guide for more information.

- **Write out the data as a MAT-file** using the `save` command, and then write a program in Fortran or C to translate the MAT-file into the desired format. See the MATLAB Application Program Interface Guide for more information.

### Delimiter-Separated Text Files

The functions `dlmread` and `dlmwrite` let you read and write delimiter-separated values from an ASCII data file. A delimiter is any character that separates the file's values. These functions are also useful for reading or writing into a specific MATLAB variable name.

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`,

```
A = dlmread('ph.dat', ';');
```
The second argument to `dlmread` specifies the delimiter, which in the previous example is a semicolon. In addition to the delimiter you specify, `dlmread` also interprets all whitespace characters as delimiters. 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
```

**USAGE** The first argument to `dlmread` is a filename, not a file identifier. You should not open the file with `fopen` prior to using `dlmread` or `dlmwrite`.

Similarly, `dlmwrite` writes delimiter-separated text to an external file:

```
A =
    1  2  3
    4  5  6

dlmwrite('myfile', A, ';;')
```

`myfile` now contains

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

**Exchanging Data Files Between Platforms**

It's sometimes necessary to work with MATLAB implementations on several different computer systems, or to transmit MATLAB applications to users on other systems. MATLAB applications consist of M-files, containing functions and scripts, and MAT-files, containing binary data. Both types of files can be transported directly between different computers:

- **M-files** are ASCII files consisting of ordinary text. They are machine independent. While different platforms terminate lines with various combinations of CR and LF characters, the MATLAB interpreter tolerates all possible combinations. (However, editors and other tools may not work correctly with M-files from other platforms.)
MAT-files are binary and machine dependent, but they can be transported between machines because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.

To use MATLAB across several different platforms, you need a program for exchanging both binary and ASCII data between the machines. When using these programs, be sure to transmit binary MAT-files in binary file mode, and ASCII M-files in ASCII file mode. Failure to set these modes correctly usually corrupts the data.

**The diary Command**

The `diary` command creates a diary of your MATLAB session in a disk file (excluding graphics). You can view and edit the resulting text file using any word processor. To create a file on your disk called `sept23.out` that contains all the commands you enter, as well as MATLAB's output, enter

```
diary sept23.out
```

To stop recording the session, use

```
diary off
```
The Startup M-File

At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`.

The file `matlabrc.m` which lives in the local directory, is reserved for use by The MathWorks and, on multiuser systems, by your system manager.

The file `startup.m` is for you to use. You can set default paths, define Handle Graphics defaults, or predefine variables in your workspace. For example, creating a `startup.m` with the line

```
addpath /home/me/mytools
```

adds a `tools` directory to your default search path.

On the PC and Macintosh, you should place the `startup.m` file in the folder named `local` in the `toolbox` folder.

On UNIX workstations, you should place the `startup.m` file in the directory named `matlab` off of your home directory, e.g., `~/.matlab`. 
Memory Utilization

MATLAB requires a contiguous area of memory to store each matrix. In particular, images and movies can consume large amounts of memory. In addition to the storage required for the matrix, the pixmap used to draw the image requires memory proportional to the area of the image on the screen. A color image of 500-by-500 pixels uses one MB of memory. To limit the amount of memory required for these operations, limit the size of the images you display.

Resolving Memory Errors

If you do not have a large enough chunk of memory to allocate a matrix, an out of memory error may occur even though you seem to have enough available memory. To consolidate the fragmented memory, you can use the MATLAB pack command, or you can allocate larger matrices earlier in the MATLAB session.

MATLAB's Memory Management

MATLAB uses the standard C functions malloc and free to allocate dynamic memory. These routines maintain a pool of memory that is allocated from the operating system relatively slowly. malloc and free allocate memory from this pool for MATLAB much more quickly. If the pool runs low, malloc asks the operating system for another large chunk of memory to replenish the pool.

As MATLAB releases memory, the pool can grow very large. To maintain speed, malloc and free do not return the additional memory to the operating system. These routines make the assumption that if you needed a large amount of memory once, you will need it again. A side effect of this algorithm is that once MATLAB has used a certain amount of memory, it is no longer available to other programs even if MATLAB is no longer using it. The memory in the pool only returns to the operating system when MATLAB terminates.

If you use an operating system tool such as ps on UNIX, the display indicates the total sum of the memory allocated by MATLAB plus the contents of the pool. This number can be deceiving since it indicates the highest level of memory use, which may or may not be the current usage.

On the Macintosh, the amount of memory available to MATLAB is set by you. To change the default setting of 16 MB, select the MATLAB program file and
choose **Get Info** from the **File** menu. Change the preferred size setting and close the dialog box.
License Manager Administration (UNIX)

Although your system administrator has probably taken care of the details of installing and configuring MATLAB's license manager, some information is helpful for all MATLAB users to know. This section contains some of the same information provided for the system administrator. For complete details, see the section by the same name in the MATLAB Installation Guide for UNIX.

On UNIX platforms, MATLAB uses a license manager called FLEXlm to manage the per-computer or per-user licensing. FLEXlm manages per-user licenses with a key system. Each time a user invokes MATLAB, the license manager considers that one key in use. When the total number of licensed keys are in use, no more users can invoke MATLAB.

FLEXlm consists of a license daemon and a product daemon that run on a server node. On UNIX computers, the server node is usually the file server on which MATLAB is installed. Throughout this section, references to the matlab directory refer to the directory where the contents of the MATLAB CD-ROM is installed.

The license and product daemons run in the background on the server node. They are responsible for checking in and out licenses as users invoke and quit MATLAB.

License Administration Tools
A number of license administration tools are available in matlab/etc, including

\begin{itemize}
\item \texttt{lmdown} Shut down all license daemons.
\item \texttt{lmhostid} Display hostid of the machine on which you are running.
\item \texttt{lmstat} Show the current status of all network licensing activities. The command \texttt{lmstat -a} displays all information. Use the switch \texttt{-f} instead of \texttt{-a} to display a list of who is using what features. \texttt{lmstat -h} displays usage help.
\item \texttt{lmstart} Start license daemons. (If license daemons are already running, you must first use \texttt{lmdown} to shut them down.)
\end{itemize}
Understanding the License File

The ASCII License File license.dat contains the details of your license, such as the number of keys you have for MATLAB, the toolboxes that you purchased, and the hostids of the licensed CPUs. If you upgrade your license or need to move the license server to a different machine, The MathWorks can give you new information by e-mail, or telephone.

Your system administrator should edit the License File to reflect your licensing information. If you edit the file yourself, follow the instructions in the Installation Guide, or run lmdown followed by lmstart. To run lmdown, you must be a member of the UNIX group lmadmin, or a member of group 0 if lmadmin does not exist.

The matlab script in matlab/bin sets the environment variable LM_LICENSE_FILE to contain the pathname where license.dat is stored. This pathname is normally matlab/etc/license.dat wherever MATLAB is stored. If necessary, you can change this environment variable to point to some other location.

The file /usr/tmp/lm_TMW5.log, where the license daemon’s output usually is redirected, contains a log of all license check-outs, check-ins, and denials. A new entry is recorded in the log each time a transaction occurs. To save file space, you can delete it occasionally.

Creating a Local Options File

You can instruct the license manager to:

- Reserve one or more keys for a particular user, group of users, or host.
- Specify the users, groups of users, or hosts that have permission to access one or more products.

To use these options, you can create a local options file and list its pathname as the fourth field on the DAEMON line in the license.dat file. Depending on the length of your path, this line may get fairly long. In the following example, this line is shown on two lines; however, you should keep it all on one line:

```
DAEMON MLM /usr/local/matlab/etc/lm_matlab
/usr/local/matlab/etc/local.options
```

A local options file is not required. If it does exist, it can have one line or many lines, reflecting your special needs. The license manager allocates keys according to these options until all keys are in use. If you try to reserve more
than the authorized number of keys in the options file, a warning message appears in the *license.log* file.

A local options file might look similar to this one:

```
RESERVE 1 MATLAB USER patricia
RESERVE 3 MATLAB HOST pegasus
RESERVE 1 CONTROL GROUP devels

INCLUDE SIGNAL HOST labrea
INCLUDE SIGNAL USER tom
EXCLUDE SIMULINK GROUP devels
GROUP devels andrea tom fred
```

The lines starting with `RESERVE` contain the number of keys for a particular product set aside for a specific user, group, or host. This does not limit the number of keys for that group or host; it simply ensures that a key will be available when you want it (unless the specified number of reserved keys has already been reached).

The lines starting with `INCLUDE` contain the products to be restricted to a particular user, group, or host; only that user, group, or host is allowed to use this product. You can have multiple `INCLUDE` lines for the same feature, including different users, groups, or hosts.

The lines starting with `EXCLUDE` contain the features to be restricted from a particular user, host, or group; that user, group, or host is not allowed to use that product. You may have multiple `EXCLUDE` lines for the same feature as well.

Any line starting with `GROUP` defines the members of a group name used in the previous lines of this file. (License manager groups are distinct from UNIX protection groups and any other groups defined outside of MATLAB.) If a group name is used in a `RESERVE`, `INCLUDE`, or `EXCLUDE` line, the group membership must be defined in a `GROUP` line.
Debugger and Profiler

MATLAB Debugger ........................................... 3-2
Debugging: An Overview ................................. 3-2
M-Files For An Example Session ..................... 3-3
Trial Run ...................................................... 3-4
Debugging on the PC ...................................... 3-5
Debugging on the Macintosh ......................... 3-10
Debugging from the Command Line ................ 3-15

M-File Profiler .............................................. 3-21
Profiling: An Overview ................................. 3-21
How the Profiler Works ................................. 3-22
The profile Command ................................. 3-22
Visualizing Profiler Results ...................... 3-25
MATLAB Debugger

The MATLAB debugger helps you identify programming errors in your MATLAB code. Using the debugger, you can view the contents of the workspace at any time during function execution, view the function call stack, and execute M-file code line by line.

MATLAB's debugger has a command line interface that is available on all platforms. The Windows and Macintosh environments also provide a visual interface to the debugger.

These sections step you through an example debugging session. Although the example M-files might be simpler than your own MATLAB code, the debugging concepts demonstrated here remain the same.

Debugging: An Overview

Debugging is the process by which you isolate and fix any problems with your code. Debugging helps to correct two kinds of errors:

- Syntax errors, such as misspelling a function name or omitting a parenthesis. MATLAB detects most syntax errors and displays a message describing the error and showing its line number in the M-file.
- Runtime errors. These errors are usually algorithmic in nature; for example, you might modify the wrong variable or perform a calculation incorrectly. Runtime errors are apparent when an M-file produces unexpected results.

You can usually correct syntax errors easily based on MATLAB's error messages. Runtime errors are more difficult to track down because the function's local workspace is lost when the error forces a return to the MATLAB base workspace. Use any of the following techniques to isolate the cause of runtime errors:

- Remove selected semicolons from the statements in your M-file. Semicolons suppress the display of intermediate calculations in the M-file. By removing the semicolons, you instruct MATLAB to display these results on your screen as the M-file executes.
- Add keyboard statements to the M-file. Keyboard statements stop M-file execution at the point where they appear and allow you to examine and change the function's local workspace. This mode is indicated by a special
prompt, “K>>.” Resume function execution by typing return and pressing the Return key.

- Comment out the leading function declaration and run the M-file as a script. This makes the intermediate results accessible in the base workspace.

- Use the MATLAB debugger. The debugger is useful for correcting runtime errors precisely because it enables you to access function workspaces and examine or change the values they contain. The debugger allows you to set and clear breakpoints, specific lines in an M-file at which execution halts. It also lets you change workspace contexts, view the function call stack, and execute the lines in an M-file one by one. This section describes these tasks in detail.

**NOTE** The M-file breakpoint information is closely associated with the copy of the M-file that MATLAB holds in memory. If you clear the M-file by editing or by issuing clear Mfile, all of Mfile’s breakpoints are also cleared.

**M-Files For An Example Session**

To try the debugger, first create an M-file called variance.m that accepts an input vector and returns an unbiased variance estimate. This file calls another M-file, sqsum, that computes the mean-removed squared sum for the input vector.

```matlab
function y = variance(x)
    mu = sum(x)/length(x);
    tot = sqsum(x, mu);
    y = tot/(length(x)-1);
```

Create the sqsum.m file exactly as it is shown below, complete with a planted bug:

```matlab
function tot = sqsum(x, mu)
    tot = 0;
    for i = 1:length(mu)
        tot = tot + ((x(i)-mu).^2);
    end
```
NOTE The example above is coded for illustrative purposes only. Whenever possible, avoid for loops and use vectorization for the most efficient execution.

**Trial Run**

Try out the M-files to see if they work correctly. Use MATLAB’s `std` function to compute results.

Create two test vectors at the command line:

```matlab
v = [1 2 3 4 5];
```

Compute the variance for each using `std`:

```matlab
var1 = std(v).^2
```

```matlab
var1 = 2.5000
```

Now try the `variance` function from above:

```matlab
myvar1 = variance(v)
```

```matlab
myvar1 = 1
```

The answer is wrong. Let’s use the debugger to isolate the error in the M-files. The following sections show example sessions on the PC and Macintosh, as well as from the command line.
### Debugging on the PC

To start debugging,

- If you’ve just created the M-files using the Editor/Debugger window, you can continue from this point.
- If you’ve created the M-files using an external text editor, click the **Open M-file** button on the Command Window to open the file in the Editor/Debugger.

The PC Editor/Debugger toolbar contains a series of debugging icons

![Debugging Icons](image-url)

whose purpose is explained in the table below:

<table>
<thead>
<tr>
<th>Toolbar Button</th>
<th>Description</th>
<th>Equivalent Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Set/Clear Breakpoint" /></td>
<td><strong>Set/Clear Breakpoint</strong>: set or clear a breakpoint at the line containing the cursor.</td>
<td><code>dbstop</code>/<code>dbclear</code></td>
</tr>
<tr>
<td><img src="image-url" alt="Clear All Breakpoints" /></td>
<td><strong>Clear All Breakpoints</strong>: clear all breakpoints that are currently set.</td>
<td><code>dbclear all</code></td>
</tr>
<tr>
<td><img src="image-url" alt="Step In" /></td>
<td><strong>Step In</strong>: execute the current line of the M-file and if the line is a call to another function, step into that function.</td>
<td><code>dbstep in</code></td>
</tr>
<tr>
<td><img src="image-url" alt="Single Step" /></td>
<td><strong>Single Step</strong>: execute the current line of the M-file.</td>
<td><code>dbstep</code></td>
</tr>
<tr>
<td><img src="image-url" alt="Continue" /></td>
<td><strong>Continue</strong>: continue execution of M-file until completion or until another breakpoint is encountered.</td>
<td><code>dbcont</code></td>
</tr>
<tr>
<td><img src="image-url" alt="Quit Debugging" /></td>
<td><strong>Quit Debugging</strong>: exit the debugging state.</td>
<td><code>dbquit</code></td>
</tr>
</tbody>
</table>

A right-button mouse click in the Editor window produces a pop-up menu of some of the options.
Setting Breakpoints

Most debugging sessions start by setting a breakpoint. Breakpoints stop M-file execution at specified lines and allow you to view or change values in the function’s workspace before resuming execution. A breakpoint is set or cleared at the line containing the cursor. A red stop sign (●) next to a line indicates that a breakpoint is set at that line. If the line selected for a breakpoint is not a valid executable line, then the breakpoint is set at the next executable line.

NOTE The debugger’s Breakpoints menu also lets you halt M-file execution if your code generates a warning, error, or NaN or Inf value.

At the beginning of the debugging session, you’re not sure where the error in the variance function is, or even if it’s in the variance.m or sqsum.m file. A logical place to insert a breakpoint is after the computation of the mean and the mean-removed squared sum. Open variance.m and set a breakpoint at line 4:

\[ y = \frac{\text{tot}}{|\text{length}(x) - 1|}; \]

The line number is indicated at the bottom right of the status bar. Set the breakpoint by positioning the cursor in the line of text and click on the breakpoint icon in the toolbar. Alternatively, you can choose Set Breakpoint from the Debug menu, or right-click to bring up the context menu and choose Set/Clear Breakpoint.
Examining Variables

To get to the breakpoint and check the values of interest, first execute the function from the Command Window:

```matlab
variance(v)
```

When execution of an M-file pauses at a breakpoint, the yellow arrow to the left of the text (✓) shows the next line to execute. A downward yellow arrow (▼) appearing to the left of the text indicates a pause at the end of the script or function. This allows you to examine variables before returning to the calling function.

Check the values of `mu` and `tot` from the debugger. Highlight the text of each variable and right-click to bring up the context menu and choose **Evaluate Selection**. Or alternatively choose **Evaluate Selection** from the **View** menu.

Both the selection and the result are displayed in the Command Window.

```matlab
K >> mu
mu =

3

K >> tot
tot =

4
```

The problem is in the `sqsumfunction`. 
Changing Workspace Context

Use the **Stack** pull-down menu in the upper-right corner of the Debugger Window to change workspace contexts. To step out from the `variance` function and see the base workspace contents, select **Base** from the menu.

Check the workspace context using `whos` or the graphical Workspace Browser. The variables `v` and `myvar1`, as well as any other variables you may have created, show up in the listing. To return to the `variance` workspace context, select **variance** from the menu.

Stepping Through Code and Continuing Execution

Clear the breakpoint at line 4 in `variance.m` by placing the cursor on the line and selecting **Clear Breakpoint** from the **Debug** menu. (Or alternatively right-click to bring up the context menu and choose **Clear Breakpoint**).

Continue execution of the M-file by selecting **Continue** from the **Debug** menu.

Open the `sqsum.m` file and set a breakpoint at line 4 to check both the loop indices and the computations that take place inside the loop. Run `variance` again, still using the vector `v` as input. Execution pauses at line 4 of `sqsum.m` if
we look at the variance function in the Editor/Debugger, we see the call to sqsum indicated by an arrow with a vertical line through it:

Evaluate the loop index $i$.

\[
\text{K} \gg i
\]

\[
i = 1
\]

Then select **Single Step** from the **Debug** menu to execute the next line. Evaluate the variable $\text{tot}$:

\[
\text{K} \gg \text{tot}
\]

\[
\text{tot} = 4
\]

Select **Single Step** again. sqsum only goes through the for loop once:

\[
\text{for } i = 1: \text{length}(\mu)
\]

The loop only iterates until the length of $\mu$, a scalar, rather than the length of $x$, the input vector.

Select **Quit Debugging** from the **Debug** menu to end the M-file execution.
To see if changing \texttt{tot} to its expected value produces the correct answer, clear the breakpoint from \texttt{sqsum} and set a breakpoint on line 4 of \texttt{variance.m}. Run \texttt{variance} once again.

\begin{verbatim}
    variance(v)
\end{verbatim}

Execution pauses after control returns from \texttt{sqsum} but before \texttt{variance} uses the returned value of \texttt{tot}. From the Command Window, set \texttt{tot} to its correct value of 10.

\begin{verbatim}
K» tot = 10
\end{verbatim}

\begin{verbatim}
tot = 10
\end{verbatim}

Select \textbf{Continue Execution} from the \textbf{Debug} menu, and the result is correct.

\textbf{End the Debug Session}

Select the \textbf{Exit Editor/Debugger} from the \textbf{File} menu to end the debugging session.

\textbf{Debugging on the Macintosh}

To start debugging,

- If you've opened the M-file by using the M-file editor, click on the debugger icon (\textbullet) on the editor's toolbar to open the M-file in the M-file debugger.
- Otherwise, choose \textbf{M-File Debugger} from the \textbf{Window} menu. Select a file to debug by choosing \textbf{Open} from the \textbf{File} menu.
The debugging icons on the toolbar are:

<table>
<thead>
<tr>
<th>Toolbar Button</th>
<th>Description</th>
<th>Equivalent Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Single Step" /></td>
<td>Single Step; execute the current line of the M-file.</td>
<td><code>dbstep</code></td>
</tr>
<tr>
<td><img src="image" alt="Step In" /></td>
<td>Step In; execute the current line of the M-file and if the line is a call to another function, step into that function.</td>
<td><code>dbstep in</code></td>
</tr>
<tr>
<td><img src="image" alt="Step Out" /></td>
<td>Step Out; continue execution until a return to the calling function is encountered.</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Continue" /></td>
<td>Continue; continue execution of M-file until completion or until another breakpoint is encountered.</td>
<td><code>dbcont</code></td>
</tr>
<tr>
<td><img src="image" alt="Quit Debugging" /></td>
<td>Quit Debugging; exit the debugging state.</td>
<td><code>dbquit</code></td>
</tr>
</tbody>
</table>

**Setting Breakpoints**
Most debugging sessions start by setting a breakpoint. Breakpoints stop M-file execution at specified lines and allow you to view or change values in the function's workspace before resuming execution. A breakpoint is set or cleared by clicking on the dash or red stop sign next to a line. A red stop sign (●) next to a line indicates that a breakpoint is set at that line. If the line selected for a breakpoint is not a valid executable line, then the breakpoint is set at the next executable line.

**NOTE** The debugger's Breakpoints menu also lets you halt M-file execution if your code generates a warning, error, or NaN or Inf value.

At the beginning of the debugging session, you're not sure where the error in the variance function is, or even if it's in the variance.m or sqsum.m file. A
logical place to insert a breakpoint is after the computation of the mean and the mean-removed squared sum. Open \texttt{variance.m} and set a breakpoint at line 4:

\[ y = \text{tot}/(\text{length}(x)-1); \]

The line number is displayed to the left of each line. Set the breakpoint by clicking on the dash next to the line. Alternatively, you can position the cursor in a line of text and choose \texttt{Set Breakpoint} from the \texttt{Breakpoints} menu.

\begin{description}
\item[Examining Variables]
To get to the breakpoint and check the values of interest, first execute the function from the Command Window:

\begin{verbatim}
variance(v)
\end{verbatim}

When execution of an M-file pauses at a breakpoint, the green arrow to the left of the text (\textarrow) shows the next line to execute. A yellow arrow appearing to the left of the text indicates a pause at the end of the script or function. This allows you to examine variables before returning to the calling function.

Check the values of \texttt{mu} and \texttt{tot} from the debugger. Highlight the text of each variable and press the \texttt{Return} or \texttt{Enter} key.

Both the selection and the result are displayed in the Command Window.

\begin{verbatim}
K>> mu
mu =
   3
K>> tot
tot =
   4
\end{verbatim}

The problem is in the \texttt{sqsum} function.

\begin{description}
\item[Changing Workspace Context]
Use the \texttt{Stack} pull-down menu in the upper-right corner of the Debugger Window to change workspace contexts. To step out from the \texttt{variance} function
and see the base workspace contents, select Base from the menu. Check the workspace context using

```matlab
whos
```

The variables \(v\) and \(myvar1\), as well as any other variables you may have created, show up in the listing. To return to the variance workspace context, select variance from the menu.

**Stepping Through Code and Continuing Execution**

Clear the breakpoint at line 4 in variance.m by placing the cursor on the line and selecting Clear Breakpoint from the Breakpoints menu. (Or alternatively click on the red stop sign icon next to the line). Continue execution of the M-file by selecting Go from the Debug menu.

Open the sqsum.m file and set a breakpoint at line 4 to check both the loop indices and the computations that take place inside the loop. Run variance again, still using the vector \(v\) as input. Execution pauses at line 4 of sqsum.

Evaluate the loop index \(i\).

```matlab
K>> i
```

\[ i = 1 \]

Then select Step from the Debug menu to execute the next line.

Evaluate the variable \(tot\):

```matlab
K>> tot
```

\[ tot = 4 \]

Select Step again. sqsum only goes through the for loop once:

```matlab
f or  i = 1: l ength( mu)
```

The loop only iterates until the length of \(mu\), a scalar, rather than the length of \(x\), the input vector.

Select Exit Debug Mode from the Debug menu to end the M-file execution.
To see if changing `tot` to its expected value produces the correct answer, clear the breakpoint from `sqsum` and set a breakpoint on line 4 of `variance.m`. Run `variance` once again.

```
variance(v)
```

Execution pauses after control returns from `sqsum` but before `variance` uses the returned value of `tot`. From the Command Window, set `tot` to its correct value of 10.

```
K>> tot = 10
```

```
tot = 10
```

Select **Go** from the **Debug** menu, and the result is correct.

**End the Debug Session**
Select **Exit Debug Mode** from the **Debug** menu and then click on the **M-File Debugger** window's close box to end the debugging session.
Debugging from the Command Line

The MATLAB debugging commands are a set of tools that allow you to debug your M-files from the command line. The most general form for each debugging command is shown below.

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set breakpoint.</td>
<td><code>dbstop at line_num in file_name</code></td>
</tr>
<tr>
<td>Remove breakpoint.</td>
<td><code>dbclear at line_num in file_name</code></td>
</tr>
<tr>
<td>Stop on warning error, or NaN/Inf generation</td>
<td><code>dstop if warning error nani nf infnan</code></td>
</tr>
<tr>
<td>Resume execution.</td>
<td><code>dbcont</code></td>
</tr>
<tr>
<td>List function call stack.</td>
<td><code>dbstack</code></td>
</tr>
<tr>
<td>List all breakpoints.</td>
<td><code>dbstatus file_name</code></td>
</tr>
<tr>
<td>Execute one or more lines.</td>
<td><code>dbstep nlines</code></td>
</tr>
<tr>
<td>List M-file with line numbers.</td>
<td><code>dbtype file_name</code></td>
</tr>
<tr>
<td>Change local workspace context (down).</td>
<td><code>dbdown</code></td>
</tr>
<tr>
<td>Change local workspace context (up).</td>
<td><code>dbup</code></td>
</tr>
<tr>
<td>Quit debug mode.</td>
<td><code>dbquit</code></td>
</tr>
</tbody>
</table>

See the online Help facility or the online MATLAB Function Reference for details on each of these functions.
Example Command Line Debugging Session
You can perform all debugging tasks from the command line. To follow this example, use the M-files from the beginning of this chapter.

```matlab
function y = variance(x)
    mu = sum(x)/length(x);
tot = sqsum(x, mu);
y = tot/(length(x)-1);
end
```

```matlab
function tot = sqsum(x, mu)
tot = 0;
for i = 1:length(mu)
    tot = tot + ((x(i)-mu).^2);
end
```

Setting Breakpoints
dbstop inserts a breakpoint at a specified line. The M-file halts before the line actually executes. Set breakpoints in `variance` after the computation of the mean (line 2), and after the computation of the mean-removed squared sum (line 3) using

```matlab
dbstop variance 3
dbstop variance 4
```

Stepping Through Code and Using Keyboard Mode
Take the vector `v = [1 2 3 4 5]` as example input. The expected values for the mean and the mean-removed squared sum are 3 and 10, respectively. See if you get the expected results at the breakpoints. Execute the function from the command line:

```matlab
variance(v)
```

MATLAB displays the next line to be executed, and its line number:

```
3   tot = sqsum(x, mu);
K>>
```

When execution stops at a breakpoint, you're automatically in keyboard mode, as indicated by the `K>>` prompt. At this prompt, you can enter standard MATLAB commands. When you're finished, enter `dbcont` to resume execution.

**NOTE** On some platforms, a debugging window may automatically appear when a function stops at a breakpoint.
When execution halts at the first breakpoint, use `whos` to see what variables are now in the workspace:

```
whos
```

To check the value of `mu`:

```
K>> mu

mu =

3
```

Use `dbstep` to step one line in the function. When execution stops again before line 4, check the value of `tot` to see if the mean-removed squared sum calculation matches the expected value:

```
K>> dbstep
4   y = tot/(length(x) - 1);
K>> tot

tot =

4
```

It appears the problem may be in the `sqsum` function.

**Changing Workspace Context**

Use `dbup` and `dbdown` to move between function workspaces and the base workspace. To step up from the `variance` function and see the base workspace contents, use

```
dbup
whos
```

The test variable `v`, as well as any other variables you may have created, shows up in the listing. To step back down to the `variance` workspace, use

```
dbdown
```
Displaying an M-File with Line Numbers

Without leaving keyboard mode, use `dbtype` to view `sqsum`. Set breakpoints to check both the loop indices and the computations that take place inside the loop:

\begin{verbatim}
K>> dbtype sqsum

1    function tot = sqsum(x,mu)
2    tot = 0;
3    for i = 1:length(mu)
4        tot = tot + ((x(i)-mu).^2);
5    end

K>> dbstop sqsum 4
K>> dbstop sqsum 5
\end{verbatim}

Viewing the Function Call Stack and Continuing Execution

Return from keyboard mode using `dbquit`. At the MATLAB prompt enter

```
dbclear variance
```

to clear the breakpoints from the `variance` function, while retaining the new `sqsum` breakpoints.

Run `variance` again, still using the vector `v` as input:

```
variance(v)
4        tot = tot + ((x(i)-mu).^2);
```

Use `dbstack` to view the function call stack, verifying that `variance` did call `sqsum`:

\begin{verbatim}
K>> dbstack
In Pat:Applications:V5:variance.m line 3
In Pat:Applications:V5:sqsum.m line 4
\end{verbatim}
Check the value of the loop index \( i \), then the value of \( \text{tot} \). After checking \( \text{tot} \), \text{dbstep} \) again to check the next value of \( i \):

```matlab
K>> i

i = 1

K>> dbstep
5 end
K>> tot

tot = 4

K>> dbstep
End of M file function Pat:Applications:V5:sqsum.m
```

The function only goes through the loop once, ending after the first iteration. Looking at the \texttt{for} statement,

```matlab
for i = 1:length(mu)
```

It's clear that there's a mistake in the line. The loop only iterates until the length of \( \mu \), a scalar, rather than the length of \( x \), the input vector.

To see if changing \( \text{tot} \) to its expected value produces the correct answer, enter

```matlab
K>> tot = 10

tot = 10
```

Use the \texttt{dbup} command to move up one workspace context, into the \texttt{variance} function workspace. It's clear that the \texttt{variance} function was taking the returned \( \text{tot} \) value of 4, dividing by \( \text{length}(x) - 1 \), also 4 in this case, and coming up with the incorrect answer of 1. Verify that it comes up with the
correct answer now that tot has the correct value, using dbcont to continue execution:

```matlab
K>> dbup
In workspace belonging to Pat:Applications:V5:variance.m
K>> dbcont
```

```
ans =
2.5000
```

**Ending A Debugging Session**
Use dbquit to end the debugging session and return to the base workspace.

Edit sqsum.m so that its for statement runs from 1 to length(x) rather than 1 to length(mu):

```
for i = 1:length(x)
```

Repeating the original trial run, we now get the expected results:

```
variance(v)
```

```
ans =
2.5000
```

```
variance(w)
```

```
ans =
468.3876
```
M-File Profiler

One way to improve the performance of your M-files is to profile them. MATLAB provides an M-file profiler that lets you see how much computation time each line of an M-file uses.

Profiling: An Overview

Profiling is a way of measuring where a program spends its time. A common mistake is to guess where most execution time is spent – it’s often surprising which portions of code actually require the most time. This may be because obvious speed issues are dealt with at design time, leaving you to discover unanticipated effects through measurement. One key to effective coding is to create an original implementation that is as simple as possible, then use a profiler to identify bottlenecks if speed is an issue. Premature optimization often increases code complexity unnecessarily without providing a real gain in performance.

Use a profiler to identify functions that are consuming the most time, then determine why you are calling them and look for ways to minimize their use. It’s often helpful to decide whether the number of times a particular function is called is reasonable. Because programs often have several layers, your code may not explicitly call the most expensive functions. Rather, functions within your code may be calling other, time-consuming functions that can be several layers down in the code. In this case it’s important to determine which of your functions are responsible for such calls.

Often the profiler helps to uncover problems that you can solve by:

- Avoiding unnecessary computation, which can arise from oversight.
- Changing your algorithm to avoid costly functions.
- Avoiding recomputation by storing results for future use.

The ultimate goal of profiling is to improve the computational speed of your code. Once you reach the point where most of the time is spent on calls to a small number of built-in functions, you’ve probably done as much optimization of the code as you can expect.
How the Profiler Works
The `profile` command allows you to specify an M-file you want to test. You can profile only one M-file at a time. Whenever the M-file executes, the profiler counts how many 0.01 second time intervals each line uses. The profiler works cumulatively - that is, it keeps adding to the line count each time the M-file executes until you manually clear the interval counts.

The profile Command
The `profile` command lets you access the profiler functionality through a simple command line interface. Its form is

```
profile keyword
```

where `keyword` can be one of

- `function_name` to start profiling for the specified function
- `on`, `off`, `done`, and `reset` to control the profiler's operation
- `report` to display a summary report for the M-file currently being profiled
- `plot` to display a Pareto plot of the profile count

Specifying the File to Profile
The `profile` command lets you specify the name of an M-file to profile. This step automatically starts profiling for that function - there is no need to turn the profiler on unless you disable profiling and want to start it again.

To specify an M-file or built-in function to profile, use

```
profile function_name
```

where `function_name` can include path specifiers. For an M-file, the profiler determines the number of lines in the file and creates a corresponding number of “bins.” Whenever the M-file executes, the profiler updates the count for a given line for each 0.01 second interval that elapses.

Turning Profiling On and Off
The `on` and `off` keywords start and stop profiling. Note that specifying a file, as described in the previous section, automatically turns profiling on. The `on` and `off` keywords allow you to enable and disable profiling in the middle of a
session. If you attempt to start profiling before you have specified a file, the profiler returns an error.

**Obtaining Profiler Results**

The `report` keyword generates a display of current profiler results. The report includes the total time spent in the function, total time, and percentage of time spent on each line, and a listing of the lines that required the most time.
An Example Profiler Session

To try out the profiler,

1. Specify a file to profile:
   ```matlab
   profile hilb
   ```
   
   hilb.m is an M-file that generates a Hilbert matrix. To see the M-file code for it, enter
   ```matlab
   type hilb
   ```

2. Execute the hilb M-file:
   ```matlab
   H = hilb(400);
   ```

3. Generate a report on the function’s execution:
   ```matlab
   profile report
   ```
   
   The profiler responds with a report such as
   ```plaintext
   Total time in "hilb": 0.14 seconds
   100% of the total time was spent on lines:
   [23 21 20 22]
   ```
   ```plaintext
   19: J = 1:n;
   0.04s, 14%  20: J = J(ones(n,1),:);
   0.07s, 24%  21: I = J';
   0.03s, 10%  22: E = ones(n,n);
   0.15s, 52%  23: H = E./(I+J-1);
   24:
   ```

4. End profiling:
   ```matlab
   profile done
   ```
   
   The exact results you see will depend on your system.
NOTE In addition to the profile command, you can access the M-file profiler through several Root properties. See the online MATLAB Function Reference for details on these properties.

Visualizing Profiler Results
The pareto function provides an easy way to visualize the profiler’s results. When used with the profiler output, pareto produces a graph that shows the values for the most time-consuming lines. For example, compare the results of the report summary and the pareto chart for the MATLAB erf function (exact results depend on your system):

```matlab
profile erfcore
z = erf(0:.01:100);
profile report
Total time in "erfcore": 5 seconds

78% of the total time was spent on lines:
[ 20 99 48 77 92 91 95 94 108 25]

19:     end
1.34s, 27%  20:     result = NaN*ones(size(x));
21:     %

24:     xbreak = 0.46875;
0.18s,  4%  25:     k = find(abs(x) <= xbreak);
26:     if any(k)

47:     %
0.39s,  8%  48:     k = find((abs(x) > xbreak) & (abs(x) <= 4.));
49:     if any(k)

76:     %
0.24s,  5%  77:     k = find(abs(x) > 4.0);
78:     if ~isempty(k)
```
for i = 1: 4
  xnum = (xnum + p(i)) .* z;
  xden = (xden + q(i)) .* z;
end
result(k) = z .* (xnum+p(5))./(xden+q(5));
result(k) = (1/sqrt(pi)–result(k))./y;
if jint ~= 2
  del = (y-z).*(y+z);
  result(k) = exp(-z.*z) .* ...
  exp(-del) .* result(k)
  k = find(~finite(result));
  ....
if jint == 0
  k = find(x > xbreak);
  result(k) = (0.5 – result(k)) + 0.5;
Now obtain profiler output to pass to pareto by calling profile with no input arguments:
  t = profile
The output in `t` is a structure containing the profiler’s results, with the `count` field containing the vector of sample counts. To see this data using `pareto`, enter

```
pareto(t.count)
```

The lines that consume the most time are shown along the bottom of the graph. The left-hand axis shows actual execution time in hundredths of a second; the right-hand axis shows percentage of total function execution time. The solid line shows cumulative execution time.
# Matrices and Linear Algebra

- Matrices and Linear Algebra .................................. 4-2
- Matrices in MATLAB ............................................ 4-4
  - Addition and Subtraction .................................. 4-6
  - Vector Products and Transpose .......................... 4-7
  - Matrix Multiplication ..................................... 4-8
  - The Identity Matrix ....................................... 4-11
  - Vector and Matrix Norms ................................. 4-12
- Solving Linear Equations .................................... 4-14
  - Square Systems ........................................... 4-16
  - Overdetermined Systems .................................. 4-17
  - Undetermined Systems .................................... 4-19
- Inverses and Determinants .................................. 4-22
  - Pseudoinverses ........................................... 4-23
- LU, QR, and Cholesky Factorizations ....................... 4-26
  - Cholesky Factorization .................................. 4-26
  - LU Factorization ......................................... 4-27
  - QR Factorization ........................................ 4-29
- Matrix Powers and Exponentials ............................. 4-33
- Eigenvalues .................................................. 4-36
- Singular Value Decomposition ............................... 4-40
Matrices and Linear Algebra

A matrix is a two-dimensional array of real or complex numbers. Linear algebra defines many matrix operations that are directly supported by MATLAB. Matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations are included.

The linear algebra functions are located in the `matfun` directory in the MATLAB toolbox.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix analysis</td>
<td>norm</td>
<td>Matrix or vector norm.</td>
</tr>
<tr>
<td></td>
<td>normest</td>
<td>Estimate the matrix 2-norm.</td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>Matrix rank.</td>
</tr>
<tr>
<td></td>
<td>det</td>
<td>Determinant.</td>
</tr>
<tr>
<td></td>
<td>trace</td>
<td>Sum of diagonal elements.</td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>Null space.</td>
</tr>
<tr>
<td></td>
<td>orth</td>
<td>Orthogonalization.</td>
</tr>
<tr>
<td></td>
<td>rref</td>
<td>Reduced row echelon form.</td>
</tr>
<tr>
<td></td>
<td>subspace</td>
<td>Angle between two subspaces.</td>
</tr>
<tr>
<td>Linear equations</td>
<td>\ and /</td>
<td>Linear equation solution.</td>
</tr>
<tr>
<td></td>
<td>inv</td>
<td>Matrix inverse.</td>
</tr>
<tr>
<td></td>
<td>cond</td>
<td>Condition number for inversion.</td>
</tr>
<tr>
<td></td>
<td>condest</td>
<td>1-norm condition number estimate.</td>
</tr>
<tr>
<td></td>
<td>chol</td>
<td>Cholesky factorization.</td>
</tr>
<tr>
<td></td>
<td>cholinc</td>
<td>Incomplete Cholesky factorization.</td>
</tr>
<tr>
<td></td>
<td>lu</td>
<td>LU factorization.</td>
</tr>
<tr>
<td></td>
<td>luinc</td>
<td>Incomplete LU factorization.</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>qr</td>
<td>Orthogonal-triangular decomposition.</td>
</tr>
<tr>
<td></td>
<td>nnls</td>
<td>Nonnegative least-squares.</td>
</tr>
<tr>
<td></td>
<td>pinv</td>
<td>Pseudoinverse.</td>
</tr>
<tr>
<td></td>
<td>lscov</td>
<td>Least squares with known covariance.</td>
</tr>
<tr>
<td>Eigenvalues and singular</td>
<td>ei g</td>
<td>Eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>values</td>
<td>svd</td>
<td>Singular value decomposition.</td>
</tr>
<tr>
<td></td>
<td>eig</td>
<td>A few eigenvalues.</td>
</tr>
<tr>
<td></td>
<td>svds</td>
<td>A few singular values.</td>
</tr>
<tr>
<td></td>
<td>poly</td>
<td>Characteristic polynomial.</td>
</tr>
<tr>
<td></td>
<td>polyeig</td>
<td>Polynomial eigenvalue problem</td>
</tr>
<tr>
<td></td>
<td>condeig</td>
<td>Condition number for eigenvalues.</td>
</tr>
<tr>
<td></td>
<td>hess</td>
<td>Hessenberg form.</td>
</tr>
<tr>
<td></td>
<td>qz</td>
<td>QZ factorization.</td>
</tr>
<tr>
<td></td>
<td>schur</td>
<td>Schur decomposition.</td>
</tr>
<tr>
<td>Matrix functions</td>
<td>expm</td>
<td>Matrix exponential.</td>
</tr>
<tr>
<td></td>
<td>logm</td>
<td>Matrix logarithm.</td>
</tr>
<tr>
<td></td>
<td>sqrtm</td>
<td>Matrix square root.</td>
</tr>
<tr>
<td></td>
<td>funm</td>
<td>Evaluate general matrix function.</td>
</tr>
</tbody>
</table>
Matrices in MATLAB

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields. (The Symbolic Math Toolboxes extend MATLAB’s capabilities to operations on various types of nonnumeric matrices.)

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric.

\[
A = \text{pascal}(3)
\]

\[
A =
\begin{bmatrix}
1 & 1 & 1 \\
1 & 2 & 3 \\
1 & 3 & 6
\end{bmatrix}
\]

The second example is not symmetric.

\[
B = \text{magic}(3)
\]

\[
B =
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]

Another example is a 3-by-2 rectangular matrix of random integers.

\[
C = \text{fix}(10*\text{rand}(3,2))
\]

\[
C =
\begin{bmatrix}
9 & 4 \\
2 & 8 \\
6 & 7
\end{bmatrix}
\]

A column vector is an m-by-1 matrix, a row vector is a 1-by-n matrix and a
scalar is a 1-by-1 matrix. The statements

\[ u = [3; 1; 4] \]
\[ v = [2 0 -1] \]
\[ s = 7 \]

produce a column vector, a row vector, and a scalar.

\[ u = \]
\[ \begin{array}{c} 3 \\ 1 \\ 4 \end{array} \]

\[ v = \]
\[ \begin{array}{ccc} 2 & 0 & -1 \end{array} \]

\[ s = \]
\[ 7 \]
Addition and Subtraction

Addition and subtraction of matrices is defined just as it is for arrays, element-by-element. Adding $A$ to $B$ and then subtracting $A$ from the result recovers $B$.

$$X = A + B$$

$$X =
\begin{bmatrix}
9 & 2 & 7 \\
4 & 7 & 10 \\
5 & 12 & 8
\end{bmatrix}$$

$$Y = X - A$$

$$Y =
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}$$

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results.

$$X = A + C$$

Error using ==> +
Matrix dimensions must agree.

$$w = v + s$$

$$w =
\begin{bmatrix}
9 & 7 & 6
\end{bmatrix}$$
Vector Products and Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the inner product, or a matrix, the outer product.

\[ x = v^*u \]

\[ x = \]

\[ 2 \]

\[ X = u^*v \]

\[ X = \]

\[ \begin{bmatrix} 6 & 0 & -3 \\ 2 & 0 & -1 \\ 8 & 0 & -4 \end{bmatrix} \]

For real matrices, the transpose operation interchanges \( a_{ij} \) and \( a_{ji} \). MATLAB uses the apostrophe (or single quote) to denote transpose. Our example matrix \( A \) is symmetric, so \( A' \) is equal to \( A \). But \( B \) is not symmetric.

\[ X = B' \]

\[ X = \]

\[ \begin{bmatrix} 8 & 3 & 4 \\ 1 & 5 & 9 \\ 6 & 7 & 2 \end{bmatrix} \]

Transposition turns a row vector into a column vector.

\[ x = v' \]

\[ x = \]

\[ \begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix} \]
If \( x \) and \( y \) are both real column vectors, the product \( x * y \) is not defined, but the two products

\[
x' * y
\]

and

\[
y' * x
\]

are the same scalar. This quantity is used so frequently, it has three different names: inner product, scalar product, or dot product.

For a complex vector or matrix, \( z \), the quantity \( z' \) denotes the complex conjugate transpose. The unconjugated complex transpose is denoted by \( z.' \), in analogy with the other array operations. So if

\[
z = \begin{bmatrix} 1+2i \\ 3+4i \end{bmatrix}
\]

then \( z' \) is

\[
\begin{bmatrix} 1-2i \\ 3-4i \end{bmatrix}
\]

while \( z.' \) is

\[
\begin{bmatrix} 1+2i \\ 3+4i \end{bmatrix}
\]

For complex vectors, the two scalar products \( x' * y \) and \( y' * x \) are complex conjugates of each other and the scalar product \( x' * x \) of a complex vector with itself is real.

**Matrix Multiplication**

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product \( C = AB \) is defined when the column dimension of \( A \) is equal to the row dimension of \( B \), or when one of them is a scalar. If \( A \) is \( m \)-by-\( p \) and \( B \) is \( p \)-by-\( n \), their product \( C \) is \( m \)-by-\( n \). The product can actually be defined using MATLAB's for loops, colon notation and vector dot products.
```
for i = 1:m
    for j = 1:n
        C(i,j) = A(i,:)*B(:,j);
    end
end

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; \( AB \) is usually not equal to \( BA \).

\[
X = A*B
\]

\[
X =
\begin{bmatrix}
15 & 15 & 15 \\
26 & 38 & 26 \\
41 & 70 & 39 \\
\end{bmatrix}
\]

\[
Y = B*A
\]

\[
Y =
\begin{bmatrix}
15 & 28 & 47 \\
15 & 34 & 60 \\
15 & 28 & 43 \\
\end{bmatrix}
\]

A matrix can be multiplied on the right by a column vector and on the left by a row vector.
\[ x = A \cdot u \]
\[ x = \begin{bmatrix} 8 \\ 17 \\ 30 \end{bmatrix} \]
\[ y = v \cdot B \]
\[ y = \begin{bmatrix} 12 & -7 & 10 \end{bmatrix} \]

Rectangular matrix multiplications must satisfy the dimension compatibility conditions.

\[ X = A \cdot C \]
\[ X = \begin{bmatrix} 17 & 19 \\ 31 & 41 \\ 51 & 70 \end{bmatrix} \]
\[ Y = C \cdot A \]

Error using \( \Rightarrow \cdot \)
Inner matrix dimensions must agree.

Anything can be multiplied by a scalar.

\[ w = s \cdot v \]
\[ w = \begin{bmatrix} 14 & 0 & -7 \end{bmatrix} \]
The Identity Matrix

Generally accepted mathematical notation uses the capital letter I to denote identity matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that \( AI = A \) and \( IA = A \) whenever the dimensions are compatible. The original version of MATLAB could not use I for this purpose because it did not distinguish between upper and lowercase letters and \( i \) already served double duty as a subscript and as the complex unit. So an English language pun was introduced. The function

\[
\text{eye}(m, n)
\]

returns an \( m \)-by-\( n \) rectangular identity matrix and \( \text{eye}(n) \) returns an \( n \)-by-\( n \) square identity matrix.

The Kronecker Tensor Product

The Kronecker product, \( \text{kron}(X, Y) \), of two matrices is the larger matrix formed from all possible products of the elements of \( X \) with those of \( Y \). If \( X \) is \( m \)-by-\( n \) and \( Y \) is \( p \)-by-\( q \), then \( \text{kron}(X, Y) \) is \( mp \)-by-\( nq \). The elements are arranged in the order

\[
\begin{bmatrix}
X(1, 1)*Y & X(1, 2)*Y & \ldots & X(1, n)*Y \\
X(m, 1)*Y & X(m, 2)*Y & \ldots & X(m, n)*Y
\end{bmatrix}
\]

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if \( X \) is the 2-by-2 matrix

\[
X = \begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

and \( I = \text{eye}(2, 2) \) is the 2-by-2 identity matrix, then the two matrices

\[
\text{kron}(X, I)
\]

and

\[
\text{kron}(I, X)
\]
are
\[
\begin{bmatrix}
1 & 0 & 2 & 0 \\
0 & 1 & 0 & 2 \\
3 & 0 & 4 & 0 \\
0 & 3 & 0 & 4 \\
\end{bmatrix}
\]
and
\[
\begin{bmatrix}
1 & 2 & 0 & 0 \\
3 & 4 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 3 & 4 \\
\end{bmatrix}
\]

**Vector and Matrix Norms**

The p-norm of a vector \( \mathbf{x} \),

\[
\| \mathbf{x} \|_p = \left( \sum x_i^p \right)^{1/p}
\]

is computed by \( \text{norm}(\mathbf{x}, p) \). This is defined by any value of \( p > 1 \), but the most common values of \( p \) are 1, 2, and \( \infty \). The default value is \( p = 2 \), which corresponds to Euclidean length.

\[
\begin{bmatrix}
\text{norm}(\mathbf{v}, 1) & \text{norm}(\mathbf{v}) & \text{norm}(\mathbf{v}, \infty) \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
3.0000 & 2.2361 & 2.0000 \\
\end{bmatrix}
\]

The p-norm of a matrix \( \mathbf{A} \),

\[
\| \mathbf{A} \|_p = \max_{\mathbf{x} \neq 0} \frac{\| \mathbf{A} \mathbf{x} \|_p}{\| \mathbf{x} \|_p}
\]

can be computed for \( p = 1, 2, \) and \( \infty \) by \( \text{norm}(\mathbf{A}, p) \). Again, the default value is
p = 2.

\[ \begin{bmatrix} \text{norm}(C, 1) & \text{norm}(C) & \text{norm}(C, \infty) \end{bmatrix} \]

ans =

19.0000  14.8015  13.0000
Solving Linear Equations

One of the most important problems in technical computing is the solution of simultaneous linear equations. In matrix notation, this problem can be stated:

Given two matrices $A$ and $B$, does there exist a unique matrix $X$ so that

$$AX = B$$

or

$$XA = B$$

It is instructive to consider a 1-by-1 example.

Does the equation

$$7x = 21$$

have a unique solution?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by division:

$$x = \frac{21}{7} = 3$$

The solution is not ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857...$, and then multiplying $7^{-1}$ by 21. This would be more work and, if $7^{-1}$ is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, slash, $/$, and backslash, \\, are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix.

$$X = A\backslash B$$

denotes the solution to the matrix equation $AX = B$.

$$X = B\backslash A$$

denotes the solution to the matrix equation $XA = B$.

You can think of “dividing” both sides of the equation $AX = B$ or $XA = B$ by $A$. The coefficient matrix $A$ is always in the “denominator”.

The dimension compatibility conditions for $X = A\backslash B$ require the two matrices $A$ and $B$ to have the same number of rows. The solution $X$ then has the same
number of columns as \( B \) and its row dimension is equal to the column dimension of \( A \). For \( X = B/A \), the roles of rows and columns are interchanged.

In practice, linear equations of the form \( AX = B \) occur more frequently than those of the form \( XA = B \). Consequently, backslash is used far more frequently than slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity

\[
(B/A)' = (A'/B')
\]

The coefficient matrix \( A \) need not be square. If \( A \) is \( m \)-by-\( n \), there are three cases.

<table>
<thead>
<tr>
<th>( m = n )</th>
<th>Square system. Seek an exact solution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m &gt; n )</td>
<td>Overdetermined system. Find a least squares solution.</td>
</tr>
<tr>
<td>( m &lt; n )</td>
<td>Underdetermined system. Find a basic solution with at most ( m ) nonzero components.</td>
</tr>
</tbody>
</table>

The backslash operator employs different algorithms to handle different kinds of coefficient matrices. The various cases, which are diagnosed automatically by examining the coefficient matrix, include:

- Permutations of triangular matrices
- Symmetric, positive definite matrices
- Square, nonsingular matrices
- Rectangular, overdetermined systems
- Rectangular, underdetermined systems
Square Systems

The most common situation involves a square coefficient matrix $A$ and a single right-hand side column vector $b$. The solution, $x = A\backslash b$, is then the same size as $b$. For example

\[
    x = A\backslash u
\]

\[
    x = \\
    10 \\
    -12 \\
    5
\]

It can be confirmed that $A\times x$ is exactly equal to $u$.

If $A$ and $B$ are square and the same size, then $X = A\backslash B$ is also that size.

\[
    X = A\backslash B
\]

\[
    X = \\
    19 \ -3 \ -1 \\
    -17 \ 4 \ 13 \\
    6 \ 0 \ -6
\]

It can be confirmed that $A\times X$ is exactly equal to $B$.

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be `pascal(3)`, which has a determinant equal to one. A later section considers the effects of roundoff error inherent in more realistic computation.

A square matrix $A$ is singular if it does not have linearly independent columns. If $A$ is singular, the solution to $AX = B$ either does not exist, or is not unique. The backslash operator, $A\backslash B$, issues a warning if $A$ is nearly singular and raises an error condition if exact singularity is detected.
Overdetermined Systems

Overdetermined systems of simultaneous linear equations are often encountered in various kinds of curve fitting to experimental data. Here is a hypothetical example. A quantity $y$ is measured at several different values of time, $t$, to produce the following observations:

<table>
<thead>
<tr>
<th>$t$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.82</td>
</tr>
<tr>
<td>0.3</td>
<td>0.72</td>
</tr>
<tr>
<td>0.8</td>
<td>0.63</td>
</tr>
<tr>
<td>1.1</td>
<td>0.60</td>
</tr>
<tr>
<td>1.6</td>
<td>0.55</td>
</tr>
<tr>
<td>2.3</td>
<td>0.50</td>
</tr>
</tbody>
</table>

This data can be entered into MATLAB with the statements

```matlab
E = [ones(size(t)) exp(-t)]
```

It is believed that the data can be modeled with a decaying exponential function.

$$y(t) = c_1 + c_2 e^{-t}$$

This equation says that the vector $y$ should be approximated by a linear combination of two other vectors, one the constant vector containing all ones and the other the vector with components $e^{-t}$. The unknown coefficients, $c_1$ and $c_2$, can be computed by doing a least squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by the 6-by-2 matrix.

$$E = [\text{ones(size}(t)) \exp(-t)]$$

$$E =$$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7408</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.4493</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.3329</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.2019</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.1003</td>
</tr>
</tbody>
</table>
The least squares solution is found with the backslash operator.

\[ c = E \backslash y \]

\[
\begin{bmatrix}
0.4760 \\
0.3413
\end{bmatrix}
\]

In other words, the least squares fit to the data is

\[ y(t) = 0.4760 + 0.3413 \ e^{-t} \]

The following statements evaluate the model at regularly spaced increments in \( t \), and then plot the result, together with the original data.

\[
T = (0:0.1:2.5)';
Y = [ones(size(T)) \ exp(-T)]*c;
plot(T,Y,'-','t,y','o');
\]

You can see that \( E \cdot c \) is not exactly equal to \( y \), but that the difference might well be less than measurement errors in the original data.

A rectangular matrix \( A \) is rank deficient if it does not have linearly independent columns. If \( A \) is rank deficient, the least squares solution to \( AX = B \) is not unique. The backslash operator, \( A \backslash B \), issues a warning if \( A \) is rank deficient and produces a basic solution that has as few nonzero elements as possible.
Undetermined Systems

Underdetermined linear systems involve more unknowns than equations. When they are accompanied by additional constraints, they are the purview of linear programming. By itself, the backslash operator deals only with the unconstrained system. The solution is never unique. MATLAB finds a basic solution, which has at most m nonzero components, but even this may not be unique. The particular solution actually computed is determined by the QR factorization with column pivoting (see a later section on the QR factorization).
Here is a small, random example.

```matlab
R = fix(10*rand(2,4))
R =
   6     8     7     3
   3     5     4     1

b = fix(10*rand(2,1))
b =
   1
   2

The linear system \(Rx = b\) involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to display the solution in rational format. The particular solution is obtained with

```matlab
format rat
p = R\b
p =
   0
   5/7
   0
-11/7
```

One of the nonzero components is \(p(2)\) because \(R(:, 2)\) is the column of \(R\) with largest norm. The other nonzero component is \(p(4)\) because \(R(:, 4)\) dominates after \(R(:, 2)\) is eliminated.
The complete solution to the overdetermined system can be characterized by adding an arbitrary vector from the null space, which can be found using the \texttt{null} function with an option requesting a “rational” basis

\[
Z = \text{null}(R, 'r')
\]

\[
Z = \\
\begin{pmatrix}
-1/2 & -7/6 \\
-1/2 & 1/2 \\
1 & 0 \\
0 & 1
\end{pmatrix}
\]

It can be confirmed that \(A^*Z\) is zero and that any vector of the form

\[
x = p + Z^*q
\]

for an arbitrary vector \(q\) satisfies \(R^*x = b\).
Inverses and Determinants

If \( A \) is square and nonsingular, the equations \( AX = I \) and \( XA = I \) have the same solution, \( X \). This solution is called the inverse of \( A \), denoted by \( A^{-1} \), and is computed by the function \( \text{inv} \). The determinant of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and roundoff error properties make it far less satisfactory for numeric computation. Nevertheless, the function \( \text{det} \) computes the determinant of a square matrix.

\[
d = \text{det}(A) \\
X = \text{inv}(A)
\]

\[
d = 1 \\
X = \begin{bmatrix}
3 & -3 & 1 \\
-3 & 5 & -2 \\
1 & -2 & 1
\end{bmatrix}
\]

Again, because \( A \) is symmetric, has integer elements, and has determinant equal to one, so does its inverse. On the other hand,

\[
d = \text{det}(B) \\
X = \text{inv}(B)
\]

\[
d = -360 \\
X = \begin{bmatrix}
0.1472 & -0.1444 & 0.0639 \\
-0.0611 & 0.0222 & 0.1056 \\
-0.0194 & 0.1889 & -0.1028
\end{bmatrix}
\]

Closer examination of the elements of \( X \), or use of \( \text{format rat} \), would reveal that they are integers divided by 360.
If A is square and nonsingular, then without roundoff error, \( X = \text{inv}(A) \cdot B \) would theoretically be the same as \( X = A \backslash B \) and \( Y = B \cdot \text{inv}(A) \) would theoretically be the same as \( Y = B / A \). But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error detection properties.

### Pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations \( AX = I \) and \( XA = I \) does not have a solution. A partial replacement for the inverse is provided by the Moore-Penrose pseudoinverse, which is computed by the \text{pinv} function.

\[
X = \text{pinv}(C)
\]

\[
0.1159 \quad -0.0729 \quad 0.0171 \\
-0.0534 \quad 0.1152 \quad 0.0418
\]

The matrix

\[
Q = X^*C
\]

\[
1.0000 \quad 0.0000 \\
0.0000 \quad 1.0000
\]

is the 2-by-2 identity, but the matrix

\[
P = C^*X
\]

\[
0.8293 \quad -0.1958 \quad 0.3213 \\
-0.1958 \quad 0.7754 \quad 0.3685 \\
0.3213 \quad 0.3685 \quad 0.3952
\]

is not the 3-by-3 identity. However, \( P \) acts like an identity on a portion of the space in the sense that \( P \) is symmetric, \( P^*C \) is equal to \( C \) and \( X^*P \) is equal to \( X \).
If \( A \) is \( m \)-by-\( n \) with \( m > n \) and full rank \( n \), then each of the three statements
\[
\begin{align*}
  x &= A \backslash b \\
  x &= \text{pinv}(A) \ast b \\
  x &= \text{inv}(A' \ast A) \ast A' \ast b
\end{align*}
\]
theoretically computes the same least squares solution \( x \), although the backslash operator does it faster.

However, if \( A \) does not have full rank, the solution to the least squares problem is not unique. There are many vectors \( x \) that minimize
\[
\text{norm}(A \ast x - b)
\]
The solution computed by \( x = A \backslash b \) is a basic solution; it has at most \( r \) nonzero components, where \( r \) is the rank of \( A \). The solution computed by \( x = \text{pinv}(A) \ast b \) is the minimal norm solution; it also minimizes \( \text{norm}(x) \). An attempt to compute a solution with \( x = \text{inv}(A' \ast A) \ast A' \ast b \) fails because \( A' \ast A \) is singular.

Here is an example to illustrates the various solutions.

\[
A = \begin{bmatrix} 1 & 2 & 3 \\
                 4 & 5 & 6 \\
                 7 & 8 & 9 \\
                 10 & 11 & 12 \end{bmatrix}
\]
does not have full rank. Its second column is the average of the first and third columns. If
\[
b = A(:, 2)
\]
is the second column, then an obvious solution to \( A \ast x = b \) is \( x = [0 \ 1 \ 0]' \). But none of the approaches computes that \( x \). The backslash operator gives
\[
\begin{align*}
  x &= A \backslash b \\
  \text{Warning: Rank deficient, rank = 2.}
\end{align*}
\]
\[
x = \begin{bmatrix} 0.5000 \\
                0 \\
                0.5000 \end{bmatrix}
\]
This solution has two nonzero components. The pseudoinverse approach gives
\[ y = \text{pinv}(A) \cdot b \]

\[ y = \]
\[ 0.3333 \]
\[ 0.3333 \]
\[ 0.3333 \]

There is no warning about rank deficiency. But \( \text{norm}(y) = 0.5774 \) is less than \( \text{norm}(x) = 0.7071 \). Finally

\[ z = \text{inv}(A' \cdot A) \cdot A' \cdot b \]

fails completely.

Warning: Matrix is singular to working precision.

\[ z = \]
\[ \text{Inf} \]
\[ \text{Inf} \]
\[ \text{Inf} \]
LU, QR, and Cholesky Factorizations

MATLAB’s linear equation capabilities are based on three basic matrix factorizations.

- Cholesky factorization for symmetric, positive definite matrices
- Gaussian elimination for general square matrices
- Orthogonalization for rectangular matrices

These three factorizations are available through the `chol`, `lu`, and `qr` functions.

All three of these factorizations make use of triangular matrices where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either forward or back substitution.

**Cholesky Factorization**

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose.

\[ A = R' R \]

where \( R \) is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be positive definite. This implies that all the diagonal elements of \( A \) are positive and that the offdiagonal elements are “not too big.” The Pascal matrices provide an interesting example. Throughout this chapter, our example matrix \( A \) has been the 3-by-3 Pascal matrix. Let’s temporarily switch to the 6-by-6.

\[ A = pascal(6) \]

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 3 & 6 & 10 & 15 & 21 \\
1 & 4 & 10 & 20 & 35 & 56 \\
1 & 5 & 15 & 35 & 70 & 126 \\
1 & 6 & 21 & 56 & 126 & 252 \\
\end{bmatrix}
\]
The elements of $A$ are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

$$R = \text{chol}(A)$$

$$R =
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 & 4 & 5 \\
0 & 0 & 1 & 3 & 6 & 10 \\
0 & 0 & 0 & 1 & 4 & 10 \\
0 & 0 & 0 & 0 & 1 & 5 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

The elements are again binomial coefficients. The fact that $R^*R$ is equal to $A$ demonstrates an identity involving sums of products of binomial coefficients.

The Cholesky factorization also applies to complex matrices. Any complex matrix which has a Cholesky factorization satisfies $A^* = A$ and is said to be Hermitian positive definite.

The Cholesky factorization allows the linear system

$$A \times x = b$$

to be replaced by

$$R^* R \times x = b$$

Because the backslash operator recognizes triangular systems, this can be solved quickly with

$$x = R \backslash (R^\top \backslash b)$$

If $A$ is $n$-by-$n$, the computational complexity of $\text{chol}(A)$ is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

**LU Factorization**

Gaussian elimination, or LU factorization, expresses any square matrix as the product of a permutation of a lower triangular matrix and an upper triangular matrix.

$$A = L \ U$$
where \(L\) is a permutation of a lower triangular matrix with ones on its diagonal and \(U\) is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix

\[
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\]

cannot be expressed as the product of triangular matrices without interchanging its two rows. Although the matrix

\[
\begin{bmatrix}
e & 1 \\
1 & 0
\end{bmatrix}
\]

can be expressed as the product of triangular matrices, when \(\epsilon\) is small the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. Partial pivoting ensures that the elements of \(L\) are bounded by one in magnitude and that the elements of \(U\) are not much larger than those of \(A\).

For example

\[
[L, U] = lu(B)
\]

\[
L =
\begin{bmatrix}
1.0000 & 0 & 0 \\
0.3750 & 0.5441 & 1.0000 \\
0.5000 & 1.0000 & 0
\end{bmatrix}
\]

\[
U =
\begin{bmatrix}
8.0000 & 1.0000 & 6.0000 \\
0 & 8.5000 & -1.0000 \\
0 & 0 & 5.2941
\end{bmatrix}
\]
The LU factorization of $A$ allows the linear system

$$A x = b$$

to be solved quickly with

$$x = U \backslash (L \backslash b)$$

Determinants and inverses are computed from the LU factorization using

$$\det(A) = \det(L) \ast \det(U) = \pm \prod \text{diag}(U)$$

and

$$\text{inv}(A) = \text{inv}(U) \ast \text{inv}(L)$$

**QR Factorization**

An orthogonal matrix, or a matrix with orthonormal columns, is a real matrix whose columns all have unit length and are perpendicular to each other. If $Q$ is orthogonal, then

$$Q' Q = I$$

The simplest orthogonal matrices are two-dimensional coordinate rotations

$$\begin{bmatrix}
\cos(\theta) & \sin(\theta) \\
-\sin(\theta) & \cos(\theta)
\end{bmatrix}$$

For complex matrices, the corresponding term is unitary. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation may also be involved.

$$A = Q R$$

or

$$A P = Q R$$

where $Q$ is orthogonal or unitary, $R$ is upper triangular, and $P$ is a permutation.
There are four variants of the QR factorization—full or economy size and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is \(m\)-by-\(n\) with \(m > n\). The full size QR factorization produces a square, \(m\)-by-\(m\) orthogonal \(Q\) and a rectangular \(m\)-by-\(n\) upper triangular \(R\).

\[
[Q, R] = qr(C)
\]

\[
Q =
\begin{bmatrix}
-0.8182 & 0.3999 & -0.4131 \\
-0.1818 & -0.8616 & -0.4739 \\
-0.5455 & -0.3126 & 0.7777
\end{bmatrix}
\]

\[
R =
\begin{bmatrix}
-11.0000 & -8.5455 \\\n0 & -7.4817 \\
0 & 0
\end{bmatrix}
\]

In many cases, the last \(m - n\) columns of \(Q\) are not needed because they are multiplied by the zeros in the bottom portion of \(R\). So the economy size QR factorization produces a rectangular, \(m\)-by-\(n\) with orthonormal columns and a square \(n\)-by-\(n\) upper triangular \(R\). For our 3-by-2 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important.

\[
[Q, R] = qr(C, 0)
\]

\[
Q =
\begin{bmatrix}
-0.8182 & 0.3999 \\
-0.1818 & -0.8616 \\
-0.5455 & -0.3126
\end{bmatrix}
\]

\[
R =
\begin{bmatrix}
-11.0000 & -8.5455 \\
0 & -7.4817
\end{bmatrix}
\]
In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of $R$ occur in decreasing order and that any linear dependence among the columns will almost certainly be revealed by examining these elements. For our small example, the second column of $C$ has a larger norm than the first, so the two columns are exchanged.

```
[ Q, R, P] = qr(C)
```

$Q =$

```
-0.3522    0.8398    -0.4131
-0.7044    -0.5285    -0.4739
-0.6163    0.1241    0.7777
```

$R =$

```
-11.3578   -8.2762
0           7.2460
0           0
```

$P =$

```
0   1
1   0
```

When the economy size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix.
[Q, R, p] = qr(C, 0)

Q =

-0.3522    0.8398
-0.7044   -0.5285
-0.6163    0.1241

R =

-11.3578   -8.2762
  0    7.2460

p =

2   1

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

\[ \text{norm}(A \cdot x - b) \]

is equal to

\[ \text{norm}(Q \cdot R \cdot x - b) \]

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

\[ \text{norm}(R \cdot x - y) \]

where \( y = Q \ast b \). Since the last \( m - n \) rows of \( R \) are zero, this expression breaks into two pieces

\[ \text{norm}(R(1:n, 1:n) \ast x - y(1:n)) \]

and

\[ \text{norm}(y(n+1:m)) \]

When \( A \) has full rank, it is possible to solve for \( x \) so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When \( A \) does not have full rank, the triangular structure of \( R \) makes it possible to find a basic solution to the least squares problem.
Matrix Powers and Exponentials

If \( A \) is a square matrix and \( p \) is a positive integer, then \( A^p \) multiplies \( A \) by itself \( p \) times.

\[
X = A^2
\]

\[
X = \\
\begin{bmatrix}
3 & 6 & 10 \\
6 & 14 & 25 \\
10 & 25 & 46
\end{bmatrix}
\]

If \( A \) is square and nonsingular, then \( A^{-p} \) multiplies \( \text{inv}(A) \) by itself \( p \) times.

\[
Y = B^{-3}
\]

\[
Y = \\
\begin{bmatrix}
0.0053 & -0.0068 & 0.0018 \\
-0.0034 & 0.0001 & 0.0036 \\
-0.0016 & 0.0070 & -0.0051
\end{bmatrix}
\]

Fractional powers, like \( A^{(2/3)} \), are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-element powers are obtained with \( .^\cdot \). For example

\[
X = A.^2
\]

\[
A = \\
\begin{bmatrix}
1 & 1 & 1 \\
1 & 4 & 9 \\
1 & 9 & 36
\end{bmatrix}
\]

The function \( \text{sqrtm}(A) \) computes \( A^{(1/2)} \) by a more accurate algorithm. The \( \text{min sqrtm} \) distinguishes this function from \( \text{sqrt}(A) \), which, like \( A.^{(1/2)} \), does its job element-by-element.
A system of linear, constant coefficient, ordinary differential equations can be written

\[ \frac{dx}{dt} = Ax \]

where \( x = x(t) \) is a vector of functions of \( t \) and \( A \) is a matrix independent of \( t \). The solution can be expressed in terms of the matrix exponential,

\[ x(t) = e^{tA}x(0) \]

The function

\[ \text{expm}(A) \]

computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix

\[ A = \begin{bmatrix} 0 & -6 & -1 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{bmatrix} \]

and the initial condition, \( x(0) \)

\[ x0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \]

The matrix exponential is used to compute the solution, \( x(t) \), to the differential equation at 101 points on the interval \( 0 \leq t \leq 1 \) with

\[ X = []; \]
\[ \text{for } t = 0:.01:1 \]
\[ \quad X = [ X \text{ expm}(t*A)*x0 ]; \]
\[ \text{end} \]

A three-dimensional phase plane plot obtained with

\[ \text{plot3}(X(1,:), X(2,:), X(3,:),'-o') \]

shows the solution spiraling in towards the origin. This behavior is related to
the eigenvalues of the coefficient matrix, which are discussed in the next section.
Eigenvalues

An eigenvalue and eigenvector of a square matrix $A$ are a scalar $\lambda$ and a vector $v$ that satisfy

$$Av = \lambda v$$

With the eigenvalues on the diagonal of a diagonal matrix $\Lambda$ and the corresponding eigenvectors forming the columns of a matrix $V$, we have

$$AV = V\Lambda$$

If $V$ is nonsingular, this becomes the eigenvalue decomposition

$$A = V\Lambda V^{-1}$$

A good example is provided by the coefficient matrix of the ordinary differential equation in the previous section,

$$A = \begin{bmatrix} 0 & -6 & -1 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{bmatrix}$$

The statement

$$\lambda = \text{eig}(A)$$

produces a column vector containing the eigenvalues. For this matrix, the eigenvalues are complex.

$$\lambda = \begin{bmatrix} -3.0710 \\ -2.4645 + 17.6008i \\ -2.4645 - 17.6008i \end{bmatrix}$$

The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as $t$ increases. The nonzero imaginary part of two of the eigenvalues, $\pm \omega$, contributes the oscillatory component, $\sin(\omega t)$, to the solution of the differential equation.
With two output arguments, \texttt{eig} computes the eigenvectors and stores the eigenvalues in a diagonal matrix.

\[
\begin{bmatrix}
V & D
\end{bmatrix} = \text{eig}(A)
\]

\begin{align*}
V &= \\
-0.8326 & -0.1203 + 0.2123i & -0.1203 - 0.2123i \\
-0.3553 & 0.4691 + 0.4901i & 0.4691 - 0.4901i \\
-0.4248 & 0.6249 - 0.2997i & 0.6249 + 0.2997i
\end{align*}

\begin{align*}
D &= \\
-3.0710 & 0 & 0 \\
0 & -2.4645 + 17.6008i & 0 \\
0 & 0 & -2.4645 - 17.6008i
\end{align*}

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, \(\text{norm}(v,2)\), equal to one.

The matrix \(V*D*inv(V)\), which can be written more succinctly as \(V*D/V\), is within roundoff error of \(A\). And, \(inv(V)*A*V\), or \(V*A*V\), is within roundoff error of \(D\).

Some matrices do not have an eigenvector decomposition. These matrices are defective, or not diagonalizable. For example,

\[
A = \\
\begin{bmatrix}
6 & 12 & 19 \\
-9 & -20 & -33 \\
4 & 9 & 15
\end{bmatrix}
\]

For this matrix

\[
\begin{bmatrix}
V & D
\end{bmatrix} = \text{eig}(A)
\]
produces

\[ V = \begin{bmatrix} 0.4741 & 0.4082 & -0.4082 \\ -0.8127 & -0.8165 & 0.8165 \\ 0.3386 & 0.4082 & -0.4082 \end{bmatrix} \]

\[ D = \begin{bmatrix} -1.0000 & 0 & 0 \\ 0 & 1.0000 & 0 \\ 0 & 0 & 1.0000 \end{bmatrix} \]

There is a double eigenvalue at \( \lambda = 1 \). The second and third columns of \( V \) are negatives of each other; they are merely different normalizations of the single eigenvector corresponding to \( \lambda = 1 \). For this matrix, a full set of linearly independent eigenvectors does not exist.

The optional Symbolic Math Toolbox extends MATLAB’s capabilities by connecting to Maple, a powerful computer algebra system. One of the functions provided by the toolbox computes the Jordan Canonical Form. This is appropriate for matrices like our example, which is 3-by-3 and has exactly known, integer elements.

\[ \text{[X, J]} = \text{Jordan}(A) \]

\[ X = \begin{bmatrix} -1.7500 & 1.5000 & 2.7500 \\ 3.0000 & -3.0000 & -3.0000 \\ -1.2500 & 1.5000 & 1.2500 \end{bmatrix} \]

\[ J = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \]

The Jordan Canonical Form is an important theoretical concept, but it is not a reliable computational tool for larger matrices, or for matrices whose elements are subject to roundoff errors and other uncertainties.
MATLAB’s advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the Schur decomposition,
\[ A = U S U^T \]
where \( U \) is an orthogonal matrix and \( S \) is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of \( S \), while the columns of \( U \) provide a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of our defective example is

\[
[U, S] = \text{schur}(A)
\]

\[
U = \\
\begin{bmatrix}
0.4741 & -0.6571 & 0.5861 \\
-0.8127 & -0.0706 & 0.5783 \\
0.3386 & 0.7505 & 0.5675 \\
\end{bmatrix}
\]

\[
S = \\
\begin{bmatrix}
-1.0000 & 21.3737 & 44.4161 \\
0 & 1.0081 & 0.6095 \\
0 & -0.0001 & 0.9919 \\
\end{bmatrix}
\]

The double eigenvalue is contained in the lower 2-by-2 block of \( S \).
Singular Value Decomposition

A singular value and corresponding singular vectors of a rectangular matrix $A$ are a scalar $\sigma$ and a pair of vectors $u$ and $v$ that satisfy

$$Av = \sigma u$$
$$A^T u = \sigma v$$

With the singular values on the diagonal of a diagonal matrix $\Sigma$ and the corresponding singular vectors forming the columns of two orthogonal matrices $U$ and $V$, we have

$$AV = U \Sigma$$
$$A^T U = V \Sigma$$

Since $U$ and $V$ are orthogonal, this becomes the singular value decomposition

$$A = U \Sigma V^T$$

The full singular value decomposition of an $m$-by-$n$ matrix involves an $m$-by-$m$ $U$, an $m$-by-$n$ $\Sigma$, and an $n$-by-$n$ $V$. In other words, $U$ and $V$ are both square and $\Sigma$ is the same size as $A$. If $A$ has many more rows than columns, the resulting $U$ can be quite large, but most of its columns are multiplied by zeros in $\Sigma$. In this situation, the economy sized decomposition saves both time and storage by producing an $m$-by-$n$ $U$, an $n$-by-$n$ $\Sigma$ and the same $V$.

The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. On the other hand, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If $A$ is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as $A$ departs from symmetry and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.
For the example matrix

\[ A = \begin{bmatrix} 9 & 4 \\ 6 & 8 \\ 2 & 7 \end{bmatrix} \]

the full singular value decomposition is

\[ [U, S, V] = \text{svd}(A) \]

\[ U = \begin{bmatrix} 0.6105 & -0.7174 & 0.3355 \\ 0.6646 & 0.2336 & -0.7098 \\ 0.4308 & 0.6563 & 0.6194 \end{bmatrix} \]

\[ S = \begin{bmatrix} 14.9359 & 0 \\ 0 & 5.1883 \\ 0 & 0 \end{bmatrix} \]

\[ V = \begin{bmatrix} 0.6925 & -0.7214 \\ 0.7214 & 0.6925 \end{bmatrix} \]
You can verify that $U S V'$ is equal to $A$ to within roundoff error. For this small problem, the economy size decomposition is only slightly smaller.

$$[U, S, V] = \text{svd}(A, 0)$$

$U =$

0.6105  -0.7174
0.6646  0.2336
0.4308  0.6563

$S =$

14.9359  0
0  5.1883

$V =$

0.6925  -0.7214
0.7214  0.6925

Again, $U S V'$ is equal to $A$ to within roundoff error.
Polynomials and Interpolation

Polynomials
- Representing Polynomials
- Polynomial Roots
- Characteristic Polynomials
- Polynomial Evaluation
- Convolution and Deconvolution
- Polynomial Derivatives
- Polynomial Curve Fitting
- Partial Fraction Expansion

Interpolation
- One-Dimensional Interpolation
- Two-Dimensional Interpolation
- Comparing Interpolation Methods
- Interpolation and Multidimensional Arrays
- Triangulation and Interpolation of Scattered Data
Polynomials

MATLAB provides functions for standard polynomial operations, such as polynomial roots, evaluation, and differentiation. In addition, there are functions for more advanced applications, such as curve fitting and partial fraction expansion.

The polynomial functions live in a directory called `polyfun` in the MATLAB Toolbox.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roots</td>
<td>Find polynomial roots.</td>
</tr>
<tr>
<td>poly</td>
<td>Polynomial with specified roots.</td>
</tr>
<tr>
<td>polyval</td>
<td>Polynomial evaluation.</td>
</tr>
<tr>
<td>polyvalm</td>
<td>Matrix polynomial evaluation.</td>
</tr>
<tr>
<td>residue</td>
<td>Partial-fraction expansion (residues).</td>
</tr>
<tr>
<td>polyfit</td>
<td>Polynomial curve fitting.</td>
</tr>
<tr>
<td>polyder</td>
<td>Polynomial derivative.</td>
</tr>
<tr>
<td>conv</td>
<td>Multiply polynomials.</td>
</tr>
<tr>
<td>deconv</td>
<td>Divide polynomials.</td>
</tr>
</tbody>
</table>

The Symbolic Math Toolbox contains additional specialized support for polynomial operations.

Representing Polynomials

MATLAB represents polynomials as row vectors containing coefficients ordered by descending powers. For example, consider the equation

\[ p(x) = x^3 - 2x - 5 \]

This is the celebrated example Wallis used when he first represented Newton's method to the French Academy. To enter this polynomial into MATLAB, use

\[ p = [1 0 -2 -5]; \]
Polynomial Roots
The roots function calculates the roots of a polynomial.

\[ r = \text{roots}(p) \]

\[ r = 
\begin{align*}
2.0946 \\
-1.0473 + 1.1359i \\
-1.0473 - 1.1359i \\
\end{align*} \]

By convention, MATLAB stores roots in column vectors. The function poly returns to the polynomial coefficients.

\[ p2 = \text{poly}(r) \]

\[ p2 = 
\begin{align*}
1 & 8.8818e-16 & -2 & -5 \\
\end{align*} \]

poly and roots are inverse functions, up to ordering, scaling, and roundoff error.

Characteristic Polynomials
The poly function also computes the coefficients of the characteristic polynomial of a matrix.

\[ A = \begin{bmatrix} 1 & 2 & 3 & -0.9; 5 & 1.75 & 6; 9 & 0 & 1 \end{bmatrix}; \]
\[ \text{poly}(A) \]

\[ \text{ans} = 
\begin{align*}
1 & -3.9500 & -1.8500 & -163.2750 \\
\end{align*} \]

The roots of this polynomial, computed with roots, are the characteristic roots, or eigenvalues, of the matrix A. (Use eig to compute the eigenvalues of a matrix directly.)
Polynomial Evaluation

The `polyval` function evaluates a polynomial at a specified value. To evaluate \( p \) at \( s = 5 \), use

\[
\text{polyval}(p, 5)
\]

\[
\text{ans} = 110
\]

It is also possible to evaluate a polynomial in a matrix sense. In this case \( p(s) = x^3 - 2x - 5 \) becomes \( p(X) = X^3 - 2X - 5I \), where \( X \) is a square matrix and \( I \) is the identity matrix. For example, create a square matrix \( X \) and evaluate the polynomial \( p \) at \( X \):

\[
X = [\begin{bmatrix} 2 & 4 & 5; -1 & 0 & 3; 7 & 1 & 5 \end{bmatrix}];
\]

\[
Y = \text{polyvalm}(p, X)
\]

\[
Y =
\begin{bmatrix}
377 & 179 & 439 \\
111 & 81 & 136 \\
490 & 253 & 639
\end{bmatrix}
\]

Convolution and Deconvolution

Polynomial multiplication and division correspond to the operations convolution and deconvolution. The functions `conv` and `deconv` implement these operations.

Consider the polynomials \( a(s) = s^2 + 2s + 3 \) and \( b(s) = 4s^2 + 5s + 6 \). To compute their product,

\[
a = [1 \ 2 \ 3]; \quad b = [4 \ 5 \ 6];
\]

\[
c = \text{conv}(a, b)
\]

\[
c =
\begin{bmatrix}
4 & 13 & 28 & 27 & 18
\end{bmatrix}
\]
Use deconvolution to divide $a(s)$ back out of the product:

$$[q, r] = \text{deconv}(c, a)$$

$q = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$

$r = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

**Polynomial Derivatives**

The `polyder` function computes the derivative of any polynomial. To obtain the derivative of the polynomial $p = [1 \ 0 \ -2 \ -5]$,

$$q = \text{polyder}(p)$$

$q = \begin{bmatrix} 3 & 0 & -2 \end{bmatrix}$

`polyder` also computes the derivative of the product or quotient of two polynomials. For example, create two polynomials $a$ and $b$:

$$a = [1 \ 3 \ 5];$$
$$b = [2 \ 4 \ 6];$$

Calculate the derivative of the product $a*b$ by calling `polyder` with a single output argument:

$$c = \text{polyder}(a, b)$$

$c = \begin{bmatrix} 8 & 30 & 56 & 38 \end{bmatrix}$
Calculate the derivative of the quotient $a/b$ by calling `polyder` with two output arguments:

```matlab
[q, d] = polyder(a, b)
```

```matlab
q =
    -2    -8    -2
```

```matlab
d =
    4    16    40    48    36
```

$q/d$ is the result of the operation.

**Polynomial Curve Fitting**

`polyfit` finds the coefficients of a polynomial that fits a set of data in a least-squares sense.

```matlab
p = polyfit(x, y, n)
```

$x$ and $y$ are vectors containing the $x$ and $y$ data to be fitted, and $n$ is the order of the polynomial to return. For example, consider the $x$-$y$ test data:

```matlab
x = [1 2 3 4 5]; y = [5.5 43.1 128 290.7 498.4];
```

A third order polynomial that approximately fits the data is

```matlab
p = polyfit(x, y, 3)
```

```matlab
p =
   -0.1917    31.5821   -60.3262    35.3400
```
Compute the values of the \texttt{polyfit} estimate over a finer range, and plot the estimate over the real data values for comparison.

\begin{verbatim}
x2 = 1:.1:5;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)
grid on
\end{verbatim}

To use these functions in an application example, see Chapter 6.

\textbf{Partial Fraction Expansion}

\texttt{residue} finds the partial fraction expansion of the ratio of two polynomials. This is particularly useful for applications that represent systems in transfer function form. For polynomials \( b \) and \( a \), if there are no multiple roots,

\[
\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k_s
\]
where \( r \) is a column vector of residues, \( p \) is a column vector of pole locations, and \( k \) is a row vector of direct terms. Consider the transfer function

\[
\frac{-4 + 8s^{-1}}{1 + 6s^{-1} + 8s^{-2}}
\]

\[
b = [-4 \; 8];
\]
\[
a = [1 \; 6 \; 8];
\]
\[
[r, p, k] = \text{residue} (b, a)
\]

\[
r =
\begin{bmatrix}
-12 \\
8
\end{bmatrix}
\]

\[
p =
\begin{bmatrix}
-4 \\
-2
\end{bmatrix}
\]

\[
k =
\begin{bmatrix}
\end{bmatrix}
\]

Given three input arguments \((r, p, \text{and} \; k)\), \text{residue} converts back to polynomial form:

\[
[b2, a2] = \text{residue} (r, p, k)
\]

\[
b2 =
\begin{bmatrix}
-4 & 8
\end{bmatrix}
\]

\[
a2 =
\begin{bmatrix}
1 & 6 & 8
\end{bmatrix}
\]
Interpolation

Interpolation is a process for estimating values that lie between known data points. It has important applications in areas such as signal and image processing. MATLAB provides a number of interpolation techniques that let you balance the smoothness of the data fit with speed of execution and memory usage.

The interpolation functions live in a directory called polyfun in the MATLAB toolbox.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp1</td>
<td>One-dimensional interpolation (table lookup).</td>
</tr>
<tr>
<td>interpft</td>
<td>One-dimensional interpolation using FFT method.</td>
</tr>
<tr>
<td>interp2</td>
<td>Two-dimensional interpolation (table lookup).</td>
</tr>
<tr>
<td>interp3</td>
<td>Three-dimensional interpolation (table lookup).</td>
</tr>
<tr>
<td>interpn</td>
<td>N-D interpolation (table lookup).</td>
</tr>
<tr>
<td>griddata</td>
<td>Data gridding and surface fitting.</td>
</tr>
<tr>
<td>spline</td>
<td>Cubic spline data interpolation.</td>
</tr>
</tbody>
</table>

One-Dimensional Interpolation

There are two kinds of one-dimensional interpolation in MATLAB:

- Polynomial interpolation
- FFT-based interpolation

Polynomial Interpolation

The function interp1 performs one-dimensional interpolation, an important operation for data analysis and curve fitting. This function uses polynomial techniques, fitting the supplied data with polynomial functions between data points and evaluating the appropriate function at the desired interpolation points. Its most general form is

\[ y_i = \text{interp1}(x, y, x_i, \text{method}) \]
Polynomials and Interpolation

y is a vector containing the values of a function, and x is a vector of the same length containing the points for which the values in y are given. xi is a vector containing the points at which to interpolate. method is an optional string specifying an interpolation method.

There are four different interpolation methods for one-dimensional data:

• **Nearest neighbor interpolation** (method = 'nearest'). This method sets the value of an interpolated point to the value of the nearest existing data point. It uses the same algorithm as the `round` function to determine which value to choose: values of xi with decimal portion less than 0.5 receive the preceding value; values of xi with decimal portion greater than or equal to 0.5 receive the succeeding value. Out-of-range points receive a value of NaN (Not a Number).

• **Linear interpolation** (method = 'linear'). This method fits separate functions between each pair of existing data points, and returns the value of the relevant function at the points specified by xi. This is the default method for the `interp1` function. Out-of-range points receive a value of NaN.

• **Cubic spline interpolation** (method = 'spline'). This method uses a series of functions to obtain interpolated data points, determining separate functions between each pair of existing data points. At its endpoint (an existing data point), each function has at least the same first and second derivatives as the function following it.

• **Cubic interpolation** (method = 'cubic'). This method fits a cubic function through y, and returns the value of this function at the points specified by xi. Out-of-range points receive a value of NaN.

All of these methods require that x be monotonic, that is, either always increasing or always decreasing from point to point. Each method works with non-uniformed spaced data. If x is already equally spaced, you can speed execution time by prepending an asterisk to the method string, for example, '*cubic c'.
Speed, Memory, and Smoothness Considerations
When choosing an interpolation method, keep in mind that some require more memory or longer computation time than others. However, you may need to trade off these resources to achieve the desired smoothness in the result.

• Nearest neighbor interpolation is the fastest method. However, it provides the worst results in terms of smoothness.
• Linear interpolation uses more memory than the nearest neighbor method, and requires slightly more execution time. Unlike nearest neighbor interpolation its results are continuous, but the slope changes at the vertex points.
• Cubic interpolation requires more memory and execution time than either the nearest neighbor or linear methods. However, both the interpolated data and its derivative are continuous.
• Cubic spline interpolation has the longest relative execution time, although it requires less memory than cubic interpolation. It produces the smoothest results of all the interpolation methods. You may obtain unexpected results, however, if your input data is non-uniform and some points are much closer together than others.

The relative performance of each method holds true even for interpolation of two-dimensional or multidimensional data. For a graphical comparison of interpolation methods, see “section Comparing Interpolation Methods” on page 5-13.

FFT-Based Interpolation
The function interpft performs one-dimensional interpolation using an FFT-based method. This method calculates the Fourier transform of a vector that contains the values of a periodic function. It then calculates the inverse Fourier transform using more points. Its form is

\[ y = \text{interpft}(x, n) \]

\( x \) is a vector containing the values of a periodic function, sampled at equally spaced points. \( n \) is the number of equally spaced points to return.
Two-Dimensional Interpolation

The function `interp2` performs two-dimensional interpolation, an important operation for image processing and data visualization. Its most general form is

\[
ZI = \text{interp2}(X, Y, Z, XI, YI, \text{method})
\]

\(Z\) is a rectangular array containing the values of a two-dimensional function, and \(X\) and \(Y\) are arrays of the same size containing the points for which the values in \(Z\) are given. \(XI\) and \(YI\) are matrices containing the points at which to interpolate the data. \text{method} is an optional string specifying an interpolation method.

There are three different interpolation methods for two-dimensional data:

- Nearest neighbor interpolation (\text{method} = 'nearest'). This method fits a piecewise constant surface through the data values. The value of an interpolated point is the value of the nearest point.
- Bilinear interpolation (\text{method} = 'linear'). This method fits a bilinear surface through existing data points. The value of an interpolated point is a combination of the values of the four closest points. This method is piecewise bilinear, and is faster and less memory-intensive than bicubic interpolation.
- Bicubic interpolation (\text{method} = 'cubic'). This method fits a bicubic surface through existing data points. The value of an interpolated point is a combination of the values of the sixteen closest points. This method is piecewise bicubic, and produces a much smoother surface than bilinear interpolation. This can be a key advantage for applications like image processing. Use bicubic interpolation when the interpolated data and its derivative must be continuous.

All of these methods require that \(X\) and \(Y\) be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the \text{meshgrid} function, or else be sure that the “pattern” of the points emulates the output of \text{meshgrid}. In addition, each method automatically maps the input to an equally spaced domain before interpolating. If \(X\) and \(Y\) are already equally spaced, you can speed execution time by prepending an asterisk to the method string, for example, '\*cubic'.

5 Polynomials and Interpolation
Comparing Interpolation Methods
This example compares two-dimensional interpolation methods on a 7-by-7 matrix of data.

1. Generate the peaks function at low resolution:
   \[
   [x, y] = \text{meshgrid}(-3:1:3);
   z = \text{peaks}(x,y);
   \text{surf}(x,y,z)
   \]

2. Generate a finer mesh for interpolation:
   \[
   [xi, yi] = \text{meshgrid}(-3:0.25:3);
   \]

3. Interpolate using nearest neighbor interpolation:
   \[
   zi1 = \text{interp2}(x,y,z,xi,yi,'nearest');
   \]

4. Interpolate using bilinear interpolation:
   \[
   zi2 = \text{interp2}(x,y,z,xi,yi,'bilinear');
   \]

5. Interpolate using bicubic interpolation:
   \[
   zi3 = \text{interp2}(x,y,z,xi,yi,'bicubic');
   \]
6 Compare the surface plots for the different interpolation methods:

\begin{align*}
\text{surf}(x_1, y_1, z_1) & \quad \text{nearest} \\
\text{surf}(x_2, y_2, z_2) & \quad \text{bilinear} \\
\text{surf}(x_3, y_3, z_3) & \quad \text{bicubic}
\end{align*}

7 Compare the contour plots for the different interpolation methods:

\begin{align*}
\text{contour}(x_1, y_1, z_1) & \quad \text{nearest} \\
\text{contour}(x_2, y_2, z_2) & \quad \text{bilinear} \\
\text{contour}(x_3, y_3, z_3) & \quad \text{bicubic}
\end{align*}

Notice that the bicubic method, in particular, produces smoother contours. This is not always the primary concern, however. For some applications, such as medical image processing, a method like nearest neighbor may be preferred because it doesn't generate any “new” data values.
Interpolation and Multidimensional Arrays

Several interpolation functions operate specifically on multidimensional data:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp3</td>
<td>Three-dimensional data interpolation.</td>
</tr>
<tr>
<td>interp5n</td>
<td>Multidimensional data interpolation.</td>
</tr>
<tr>
<td>ndgrid</td>
<td>Multidimensional data gridding (ndfun directory).</td>
</tr>
</tbody>
</table>

Interpolation of Three-Dimensional Data

The function `interp3` performs three-dimensional interpolation, finding interpolated values between points of a three-dimensional set of samples V. You must specify a set of known data points:

- X, Y, and Z matrices specify the points for which values of V are given.
- A matrix V contains values corresponding to the points in X, Y, and Z.

The most general form for `interp3` is

\[ V_I = \text{interp3}(X,Y,Z,V,X_I,Y_I,Z_I,\text{method}) \]

\(X_I, Y_I,\) and \(Z_I\) are the points at which `interp3` interpolates values of V. For out-of-range values, `interp3` returns NaN.

There are three different interpolation methods for three-dimensional data:

- Nearest neighbor interpolation (method = 'nearest'). This method chooses the value of the nearest point.
- Trilinear interpolation (method = 'linear'). This method uses piecewise linear interpolation based on the values of the nearest eight points.
- Tricubic interpolation (method = 'cubic'). This method uses piecewise cubic interpolation based on the values of the nearest sixty-four points.

All of these methods require that X, Y, and Z be monotonic, that is, either always increasing or always decreasing in a particular direction. In addition, you should prepare these matrices using the `ndgrid` function, or else be sure that the “pattern” of the points emulates the output of `ndgrid`. 

5-15
Each method automatically maps the input to an equally spaced domain before interpolating. If \( x \) is already equally spaced, you can speed execution time by prepending an asterisk to the \texttt{method} string, for example, \texttt{'*cubic'}.

**Interpolation of Higher-Dimensional Data**

The function \texttt{interpn} performs multidimensional interpolation, finding interpolated values between points of a multidimensional set of samples \( V \). The most general form for \texttt{interpn} is

\[
\text{VI} = \text{interpn}(X1, X2, X3, \ldots, V, Y1, Y2, Y3, \ldots, \text{method})
\]

1, 2, 3, ... are matrices that specify the points for which values of \( V \) are given. \( V \) is a matrix that contains the values corresponding to these points. 1, 2, 3, ... are the points for which \texttt{interpn} returns interpolated values of \( V \). For out-of-range values, \texttt{interpn} returns NaN.

\( Y1, Y2, Y3, \ldots \) must be either arrays of the same size, or vectors. If they are vectors of different sizes, \texttt{interpn} passes them to \texttt{ndgrid} and then uses the resulting arrays.

There are three different interpolation methods for multidimensional data:

- **Nearest neighbor interpolation** (\texttt{method} = \texttt{'nearest'}). This method chooses the value of the nearest point.
- **Linear interpolation** (\texttt{method} = \texttt{'linear'}). This method uses piecewise linear interpolation based on the values of the nearest two points in each dimension.
- **Cubic interpolation** (\texttt{method} = \texttt{'cubic'}). This method uses piecewise cubic interpolation based on the values of the nearest four points in each dimension.

All of these methods require that \( X1, X2, X3 \) be monotonic. In addition, you should prepare these matrices using the \texttt{ndgrid} function, or else be sure that the “pattern” of the points emulates the output of \texttt{ndgrid}.

Each method automatically maps the input to an equally spaced domain before interpolating. If \( x \) is already equally spaced, you can speed execution time by prepending an asterisk to the \texttt{method} string; for example, \texttt{'*cubic'}. 

---

5 Polynomials and Interpolation


**Multidimensional Data Gridding**

The `ndgrid` function generates arrays of data for multidimensional function evaluation and interpolation. `ndgrid` transforms the domain specified by a series of input vectors into a series of output arrays. The i'th dimension of these output arrays are copies of the elements of input vector $x_i$.

The syntax for `ndgrid` is

```
[X1, X2, X3, ...] = ndgrid(x1, x2, x3, ...)
```

For example, assume that you want to evaluate a function of three variables over a given range. Consider the function

$$z = x_2 e^{(-x_1^2 - x_2^2 - x_3^2)}$$

for \(-2\pi \leq x_1 \leq 0, 2\pi \leq x_2 \leq 4\pi, \) and \(0 \leq x_3 \leq 2\pi. \) To evaluate and plot this function:

```matlab
x1 = -2:0.2:2;
x2 = -2:0.25:2;
x3 = -2:0.16:2;
[X1, X2, X3] = ndgrid(x1, x2, x3);
z = X2 .* exp(-X1.^2 -X2.^2 -X3.^2);
slice(X2, X1, X3, z, [-1.2 2 2], 2, [-2 0 2])
```
Triangulation and Interpolation of Scattered Data
MATLAB provides routines that aid in the analysis of closest-point problems and geometric analysis:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delaunay</td>
<td>Delaunay triangulation.</td>
</tr>
<tr>
<td>dsearch</td>
<td>Search Delaunay triangulation for nearest point.</td>
</tr>
<tr>
<td>tsearch</td>
<td>Closest triangle search.</td>
</tr>
<tr>
<td>convhull</td>
<td>Convex hull.</td>
</tr>
<tr>
<td>voronoi</td>
<td>Voronoi diagram.</td>
</tr>
<tr>
<td>inpolygon</td>
<td>True for points inside polygonal region.</td>
</tr>
<tr>
<td>rectint</td>
<td>Area of intersection for two or more rectangles.</td>
</tr>
<tr>
<td>polyarea</td>
<td>Area of polygon.</td>
</tr>
</tbody>
</table>
Delaunay Triangulation

The `delaunay` function returns a set of triangles such that no data points are contained in any triangle's circumcircle. To try `delaunay`, load the `seamount` data set supplied with MATLAB and view the data as a simple scatter plot.

```matlab
load seamount
plot(x, y, '.', 'markersize', 12)
xlabel('Longitude'), ylabel('Latitude')
grid on
```

Apply Delaunay triangulation and overplot the resulting triangles on the scatter plot:

\[
\begin{align*}
\text{tri} &= \text{delaunay}(x,y); \\
\text{hold on, tri mesh(tri,x,y,z), hold off} \\
\text{hidden off; grid on} \\
\text{xlabel('Longitude'); ylabel('Latitude')} 
\end{align*}
\]
Here's a contour plot:

```matlab
[ xi , yi ] = meshgrid(210.8:.01:211.8,–48.5:.01:–47.9);
zi = griddata(x,y,z,xi,yi,'cubic');
[c, h] = contour(xi,yi,zi,'c–'); clabel(c,h)
```

The arguments for `meshgrid` encompass the largest and smallest x and y values in the original `seamount` data. To obtain these values, use

```matlab
min(min(x))
max(max(x))
```

and

```matlab
min(min(y))
max(max(y))
```
**Closest-Point Searches.** You can search through the Delaunay triangulation data with two functions:

- `dsearch` finds the point closest to a point you specify.
- `tsearch`, given a point `(xi, yi)`, returns an index into the `delaunay` output that specifies the enclosing triangle for the point.

**Voronoi Diagrams**

Voronoi diagrams are a closest-point plotting technique related to Delaunay triangulation. The Voronoi diagram for the `seamount` data is

```matlab
load seamount
voronoi(x, y)
grid on
```
Convex Hulls
The `convhull` function returns the indices of the points in a data set that comprise the convex hull for the set. For example, to view the convex hull for the `seamount` data:

```matlab
load seamount
plot(x, y, '.', 'markersize', 10)
k = convhull(x, y);
hold on, plot(x(k), y(k)), hold off
grid on
```
Data Analysis and Statistics

Column-Oriented Data Sets

Basic Data Analysis Functions
Covariance and Correlation Coefficients
Finite Differences

Data Pre-Processing
Missing Values
Removing Outliers

Regression and Curve Fitting
Polynomial Regression
Linear-in-the-Parameters Regression
Multiple Regression

Case Study: Curve Fitting
Polynomial Fit
Analyzing Residuals
Exponential Fit
Error Bounds

Difference Equations and Filtering

Fourier Analysis and the Fast Fourier Transform (FFT)
Magnitude and Phase of Transformed Data
FFT Length Versus Speed
This chapter introduces MATLAB’s data analysis capabilities. It discusses how to organize arrays for data analysis, how to use simple descriptive statistics functions, and how to perform data pre-processing tasks in MATLAB. It also discusses other data analysis topics, including regression, curve fitting, data filtering, and fast Fourier transforms (FFTs).

The data analysis and statistics functions are in the directory `datafun` in the MATLAB Toolbox. Use online help to get a complete list of functions.

A number of related toolboxes provide advanced functionality for specialized data analysis applications.

<table>
<thead>
<tr>
<th>Toolbox</th>
<th>Data Analysis Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>Nonlinear curve fitting and regression.</td>
</tr>
<tr>
<td>Signal Processing</td>
<td>Signal processing, filtering, and frequency analysis.</td>
</tr>
<tr>
<td>Spline</td>
<td>Curve fitting and regression.</td>
</tr>
<tr>
<td>Statistics</td>
<td>Advanced statistical analysis, nonlinear curve fitting, and regression.</td>
</tr>
<tr>
<td>System Identification</td>
<td>Parametric / ARMA modeling.</td>
</tr>
<tr>
<td>Wavelet</td>
<td>Wavelet analysis.</td>
</tr>
</tbody>
</table>
Column-Oriented Data Sets

Univariate statistical data is typically stored in individual vectors when working with MATLAB. The vectors can be either 1-by-n or n-by-1. For multivariate data, a matrix is the natural representation but there are, in principle, two possibilities for orientation. By MATLAB convention, however, the different variables are put into columns, allowing observations to vary down through the rows. Therefore, a data set consisting of twenty four samples of three variables is stored in a matrix of size 24-by-3.

Consider a sample data set comprising vehicle traffic count observations at three locations over a twenty-four hour period.

<table>
<thead>
<tr>
<th>Time</th>
<th>Location 1</th>
<th>Location 2</th>
<th>Location 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>01h00</td>
<td>11</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>02h00</td>
<td>7</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>03h00</td>
<td>14</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>04h00</td>
<td>11</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>05h00</td>
<td>43</td>
<td>51</td>
<td>69</td>
</tr>
<tr>
<td>06h00</td>
<td>38</td>
<td>46</td>
<td>76</td>
</tr>
<tr>
<td>07h00</td>
<td>61</td>
<td>132</td>
<td>186</td>
</tr>
<tr>
<td>08h00</td>
<td>75</td>
<td>135</td>
<td>180</td>
</tr>
<tr>
<td>09h00</td>
<td>38</td>
<td>88</td>
<td>115</td>
</tr>
<tr>
<td>10h00</td>
<td>28</td>
<td>36</td>
<td>55</td>
</tr>
<tr>
<td>11h00</td>
<td>12</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>12h00</td>
<td>18</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>13h00</td>
<td>18</td>
<td>19</td>
<td>29</td>
</tr>
<tr>
<td>14h00</td>
<td>17</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>15h00</td>
<td>19</td>
<td>36</td>
<td>48</td>
</tr>
<tr>
<td>Time</td>
<td>Location 1</td>
<td>Location 2</td>
<td>Location 3</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>16h00</td>
<td>32</td>
<td>47</td>
<td>10</td>
</tr>
<tr>
<td>17h00</td>
<td>42</td>
<td>65</td>
<td>92</td>
</tr>
<tr>
<td>18h00</td>
<td>57</td>
<td>66</td>
<td>151</td>
</tr>
<tr>
<td>19h00</td>
<td>44</td>
<td>55</td>
<td>90</td>
</tr>
<tr>
<td>20h00</td>
<td>114</td>
<td>145</td>
<td>257</td>
</tr>
<tr>
<td>21h00</td>
<td>35</td>
<td>58</td>
<td>68</td>
</tr>
<tr>
<td>22h00</td>
<td>11</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>23h00</td>
<td>13</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>24h00</td>
<td>10</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
The raw data is stored in the ASCII file, `count.dat`:

```
11 11 9
7 13 11
14 17 20
11 13 9
43 51 69
38 46 76
61 132 186
75 135 180
38 88 115
28 36 55
12 12 14
18 27 30
18 19 29
17 15 18
19 36 48
32 47 10
42 65 92
57 66 151
44 55 90
114 145 257
35 58 68
11 12 15
13 9 15
10 9 7
```

Use the `load` command to import the data:
```
load count.dat
```

This creates a matrix `count` in the workspace.

For this example, there are 24 observations of three variables. This is confirmed by:
```
[n, p] = size(count)
```
```
n =
   24
```
```
p =
   3
```
Create a time vector, \( t \), of integers from 1 to \( n \):
\[
t = 1:n;
\]

Now plot the counts versus time and annotate the plot:
\[
\text{set(0,'defaultaxeslinestyleorder','-|--|-.')}
\text{set(0,'defaultaxescolororder',[0 0 0])}
\text{plot(t,count), legend('Location 1','Location 2','Location 3',0)}
\text{xlabel('Time'), ylabel('Vehicle Count'), grid on}
\]

The plot shows the vehicle counts at three locations over a twenty-four hour period.
Basic Data Analysis Functions

A collection of functions provides basic column-oriented data analysis capabilities:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>Largest component.</td>
</tr>
<tr>
<td>min</td>
<td>Smallest component.</td>
</tr>
<tr>
<td>mean</td>
<td>Average or mean value.</td>
</tr>
<tr>
<td>median</td>
<td>Median value.</td>
</tr>
<tr>
<td>std</td>
<td>Standard deviation.</td>
</tr>
<tr>
<td>sort</td>
<td>Sort in ascending order.</td>
</tr>
<tr>
<td>sortrows</td>
<td>Sort rows in ascending order.</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of elements.</td>
</tr>
<tr>
<td>prod</td>
<td>Product of elements.</td>
</tr>
<tr>
<td>diff</td>
<td>Difference function and approximate derivative.</td>
</tr>
<tr>
<td>trapz</td>
<td>Trapezoidal numerical integration.</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum of elements.</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product of elements.</td>
</tr>
<tr>
<td>cumtrapz</td>
<td>Cumulative trapezoidal numerical integration.</td>
</tr>
</tbody>
</table>

For vector input arguments to these functions, it does not matter whether the vectors are oriented in row or column direction. For array arguments, however, the functions operate in a column-oriented fashion on the data in the array. This means, for example, that if you apply max to an array, the result is a row vector containing the maximum values over each column.
NOTE You can add more functions to this list using M-files, but when doing so, you must exercise care to handle the row vector case. If you are writing your own column-oriented M-files, check other M-files; for example, mean.m and diff.m

Continuing with the vehicle traffic count example, the statements

\[
\begin{align*}
\text{mx} &= \text{max}(\text{count}) \\
\text{mu} &= \text{mean}(\text{count}) \\
\text{sigma} &= \text{std}(\text{count})
\end{align*}
\]

result in

\[
\begin{align*}
\text{mx} &= \\
&= \begin{bmatrix} 114 & 145 & 257 \end{bmatrix} \\
\text{mu} &= \\
&= \begin{bmatrix} 32.0000 & 46.5417 & 65.5833 \end{bmatrix} \\
\text{sigma} &= \\
&= \begin{bmatrix} 25.3703 & 41.4057 & 68.0281 \end{bmatrix}
\end{align*}
\]

To locate the index at which the minimum or maximum occurs, a second output parameter can be specified. For example,

\[
[\text{mx, indx}] = \text{min}(\text{count})
\]

\[
\begin{align*}
\text{mx} &= \\
&= \begin{bmatrix} 7 & 9 & 7 \end{bmatrix} \\
\text{indx} &= \\
&= \begin{bmatrix} 2 & 23 & 24 \end{bmatrix}
\end{align*}
\]

shows that the lowest vehicle count is recorded at 02h00 for the first observation point (column one) and at 23h00 and 24h00 for the other observation points.
You can subtract the mean from each column of the data using an outer product involving a vector of \( n \) ones.

\[
\begin{align*}
[n, p] &= \text{size}(\text{count}) \\
e &= \text{ones}(n, 1) \\
x &= \text{count} - e*\mu
\end{align*}
\]

Rearranging the data may help you evaluate a vector function over an entire dataset. For example, to find the smallest value in the entire data set, use

\[
\text{min}(\text{count}(:))
\]

which produces:

\[
\text{ans} = 7
\]

The syntax \( \text{count}(:) \) rearranges the 24-by-3 matrix into a 72-by-1 column vector.

**Covariance and Correlation Coefficients**

MATLAB’s statistical capabilities include two functions for the computation of correlation coefficients and covariance:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| \text{cov} | Variance of vector – measure of spread or dispersion of sample variable.  
Covariance of matrix – measure of strength of linear relationships between variables. |
| \text{corrcoef} | Correlation coefficient – normalized measure of linear relationship strength between variables. |

cov returns the variance for a vector of data. The variance of the data in the first column of count is

\[
\text{cov}(\text{count}(:, 1))
\]

\[
\text{ans} = 643.6522
\]
For an array of data, \texttt{cov} calculates the covariance matrix. The variance values for the array columns are arranged along the diagonal of the covariance matrix. The remaining entries reflect the covariance between the columns of the original array. For an \( m \)-by-\( n \) matrix, the covariance matrix has size \( n \)-by-\( n \). For example, the covariance matrix for \texttt{count}, \texttt{cov(count)}, is arranged as:

\[
\begin{bmatrix}
\sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 \\
\sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 \\
\sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2
\end{bmatrix}
\]

\texttt{corrcoef} produces a matrix of correlation coefficients for an array of data where each row is an observation and each column is a variable. The correlation coefficient is a normalized measure of the strength of the linear relationship between two variables. Uncorrelated data results in a correlation coefficient of 0; equivalent data sets have a correlation coefficient of 1.

For an \( m \)-by-\( n \) matrix, the correlation coefficient matrix has size \( n \)-by-\( n \). The arrangement of the elements in the correlation coefficient matrix corresponds to the location of the elements in the covariance matrix described above.

For our traffic count example

\texttt{corrcoef(count)}

results in

\[
\begin{array}{ccc}
1.0000 & 0.9331 & 0.9599 \\
0.9331 & 1.0000 & 0.9553 \\
0.9599 & 0.9553 & 1.0000
\end{array}
\]

Clearly there is a strong linear correlation between the three traffic counts observed at the three locations, as the results are close to 1.
Finite Differences
MATLAB provides three functions for finite difference calculations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gradient</code></td>
<td>Numerical partial derivatives a matrix.</td>
</tr>
<tr>
<td><code>del2</code></td>
<td>Discrete Laplacian of a matrix.</td>
</tr>
</tbody>
</table>

The `diff` function computes the difference between successive elements in a numeric vector. That is, `diff(X)` is \([X(2)-X(1) \quad X(3)-X(2) \quad \ldots \quad X(n)-X(n-1)]\). So, for a vector \(A\),

\[
A = [9 \quad -2 \quad 3 \quad 0 \quad 1 \quad 5]\;
\]

\[
diff(A)
\]

\[
\begin{array}{c}
\text{ans} = \\
-11 \quad 5 \quad -3 \quad 1 \quad 4 \quad -1
\end{array}
\]

Besides computing the first difference, `diff` is useful for determining certain characteristics of vectors. For example, you can use `diff` to determine if a vector is monotonic (elements are always either increasing or decreasing), or if a vector has equally spaced elements. For example, given a vector \(x\),

\[
diff(x) == 0\quad \text{Tests for repeated elements.}
\]

\[
\text{all}(diff(x) > 0)\quad \text{Tests for monotonicity.}
\]

\[
\text{all}(diff(diff(x)) == 0)\quad \text{Tests for equally spaced vector elements.}
\]
Data Pre-Processing

Missing Values
The special value, NaN, stands for Not-a-Number in MATLAB. IEEE floating-point arithmetic convention specifies NaN as the result of undefined expressions like 0/0.

The correct handling of missing data is a difficult problem and often varies in different situations. For data analysis purposes, it is often convenient to use NaNs to represent missing values or data that are not available.

MATLAB treats NaNs in a uniform and rigorous way; they propagate naturally through to the final result in any calculation. Any mathematical calculation involving NaNs will produce NaNs in the results.

For example, consider a matrix containing the 3-by-3 magic square with its center element set to NaN

```matlab
a = magic(3); a(2,2) = NaN

a =
  8   1   6
  3  NaN   7
  4   9   2
```

Now sum down each column of the matrix

```matlab
sum(a)

ans =
  15  NaN  15
```

Any mathematical calculation involving NaNs propogates NaNs through to the final result as appropriate.
You should remove NaNs from the data before performing statistical computations. Here are some ways to do so:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i = find(~isnan(x));</code></td>
<td>Find indices of elements in vector that are not NaNs, then keep only the non-NaN elements.</td>
</tr>
<tr>
<td><code>x = x(i)</code></td>
<td></td>
</tr>
<tr>
<td><code>x = x(find(~isnan(x)))</code></td>
<td>Remove NaNs from vector.</td>
</tr>
<tr>
<td><code>x = x(~isnan(x));</code></td>
<td>Remove NaNs from vector (faster).</td>
</tr>
<tr>
<td><code>x(:) = [];</code></td>
<td>Remove NaNs from vector.</td>
</tr>
<tr>
<td><code>X(any(~isnan(X)'),:) = [];</code></td>
<td>Remove any rows of matrix X containing NaNs.</td>
</tr>
</tbody>
</table>

**NOTE** You must use the special function isnan to find NaNs because, by IEEE arithmetic convention, the logical comparison, NaN == NaN always produces 0. You cannot use `x(x==NaN) = []` to remove NaNs from your data.

If you frequently need to remove NaNs, write a short M-file, for example

```matlab
function X = excise(X)
    X(any(~isnan(X)'),:) = [];
end
```

Now, typing

```matlab
X = excise(X);
```

accomplishes the same thing.

**Removing Outliers**

You can remove outliers or misplaced data points from a data set in much the same manner as NaNs. For the vehicle traffic count data, the mean and standard deviations of each column of the data are

```matlab
mu = mean(count);
sigma = std(count);
```
The number of rows with outliers greater than three standard deviations is obtained with:

```matlab
[n, p] = size(count)
outliers = abs(count - mu(ones(n, 1), :) > 3*sigma(ones(n, 1), :));
nout = sum(outliers)
```

```
1 0 0
```

There is one outlier in the first column. Remove this entire observation with

```matlab
count(any(outliers', :) = [];
```
Regression and Curve Fitting

It is often useful to find functions that describe the relationship between some variables you have observed. Identification of the coefficients of the function often leads to the formulation of an overdetermined system of simultaneous linear equations. These coefficients can be efficiently found using the MATLAB backslash operator.

Suppose you measure a quantity $y$ at several values of time $t$.

```matlab
t = [0 .3 .8 1.1 1.6 2.3]';
y = [0.5 0.82 1.14 1.25 1.35 1.40]';
plot(t,y,'o'), grid on
```

Polynomial Regression

Based on the plot, it is possible that the data may be modeled by a polynomial function

$$y = a_0 + a_1 t + a_2 t^2$$

The unknown coefficients $a_0$, $a_1$, and $a_2$, can be computed by doing a least squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in three unknowns,
Data Analysis and Statistics

represented by the 6-by-3 matrix.

\[
\begin{bmatrix}
1 & t_1 & t_1^2 \\
1 & t_2 & t_2^2 \\
1 & t_3 & t_3^2 \\
1 & t_4 & t_4^2 \\
1 & t_5 & t_5^2 \\
1 & t_6 & t_6^2
\end{bmatrix}
\]

\[
X = \begin{bmatrix}
1.0000 & 0 & 0 \\
1.0000 & 0.3000 & 0.0900 \\
1.0000 & 0.8000 & 0.6400 \\
1.0000 & 1.1000 & 1.2100 \\
1.0000 & 1.6000 & 2.5600 \\
1.0000 & 2.3000 & 5.2900
\end{bmatrix}
\]

The solution is found with the backslash operator.

\[
a = X\backslash y
\]

\[
a = \begin{bmatrix}
0.5318 \\
0.9191 \\
-0.2387
\end{bmatrix}
\]

The second order polynomial model of the data is therefore

\[
y = 0.5318 + 0.9191 t - 0.2387 t^2
\]
Now evaluate the model at regularly spaced points and overlay the original data in a plot.

\[
T = (0:0.1:2.5)';
Y = [ones(size(T)) T T.^2]*a;
plot(T,Y,'-',t,y,'o'), grid on
\]

Clearly this fit does not perfectly approximate the data. We could either increase the order of the polynomial fit, or explore some other functional form to get a better approximation.

**Linear-in-the-Parameters Regression**

Instead of a polynomial function, we could try using a function that is linear-in-the-parameters. In this case, consider the exponential function

\[
y = a_0 + a_1 e^{-t} + a_2 te^{-t}
\]

The unknown coefficients \(a_0, a_1,\) and \(a_2\) are computed by performing a least squares fit. Con
and solve the set of simultaneous equations by forming the regression matrix, $X$, and solving for the coefficients using the backslash operator.

$$X = \begin{bmatrix}
\text{ones(size(t))} & \exp(-t) & t \cdot \exp(-t)
\end{bmatrix};$$

$$a = X \backslash y$$

$$a =
\begin{bmatrix}
1.3974 \\
-0.8988 \\
0.4097
\end{bmatrix}$$

The fitted model of the data is therefore

$$y = 1.3974 - 0.8988 \exp(-t) + 0.4097 t \exp(-t)$$

Now evaluate the model at regularly spaced points and overlay the original data in a plot.

$$T = (0:0.1:2.5)';$$
$$Y = \begin{bmatrix}
\text{ones(size(T))} & \exp(-T) & T \cdot \exp(-T)
\end{bmatrix} \cdot a;$$
$$\text{plot}(T,Y,'-t',y,'o'), \text{grid on}$$

This is a much better fit than the second order polynomial function.
Multiple Regression

If $y$ is a function of more than one independent variable, the matrix equations that express the relationships among the variables can be expanded to accommodate the additional data.

Suppose we measure a quantity $y$ for several values of parameters $x_1$ and $x_2$. The observations are entered as

$$
x_1 = [.2 .5 .6 .8 1.0 1.1]';
$$

$$
x_2 = [.1 .3 .4 .9 1.1 1.4]';
$$

$$
y = [.17 .26 .28 .23 .27 .24]';
$$

A multivariate model of the data is

$$
y = a_0 + a_1 x_1 + a_2 x_2
$$

Multiple regression solves for unknown coefficients $a_0$, $a_1$, and $a_2$, by performing a least squares fit. Construct and solve the set of simultaneous equations by forming the regression matrix, $X$, and solving for the coefficients using the backslash operator.

$$
X = [\text{ones(size(x1))} \ x1 \ x2];
$$

$$
a = X \backslash y
$$

$$
a = 
\begin{align*}
0.1018 \\
0.4844 \\
-0.2847
\end{align*}
$$

The least-squares fit model of the data is therefore

$$
y = 0.108 + 0.4844 \ x_1 - 0.2846 \ x_2
$$

To validate the model, find the maximum of the absolute value of the deviation of the data from the model:

$$
Y = X \ast a;
$$

$$
\text{MaxErr} = \max(\text{abs}(Y - y))
$$

$$
\text{MaxErr} = 0.0038
$$

This is sufficiently small to be confident the model reasonably fits the data.
Case Study: Curve Fitting

This section provides an overview of some of MATLAB’s basic data analysis capabilities in the form of a case study. The examples that follow work with a collection of census data, using MATLAB functions to experiment with fitting curves to the data.

The file census.mat contains US population data for the years 1790 through 1990. Load it into MATLAB:

```
load census
```

Your workspace now contains two new variables, cdate and pop.

- `cdate` is a column vector containing the years from 1790 to 1990 in increments of 10.
- `pop` is a column vector with the US population figures that correspond to the years in `cdate`.

**Polynomial Fit**

A first try in fitting the census data might be a simple polynomial fit. Two MATLAB functions help with this process:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>polyfit</code></td>
<td>Polynomial curve fit.</td>
</tr>
<tr>
<td><code>polyval</code></td>
<td>Evaluation of polynomial fit.</td>
</tr>
</tbody>
</table>

MATLAB’s `polyfit` function generates a “best fit” polynomial (in the least squares sense) of a specified order for a given set of data. For a polynomial fit of order 4:

```
p = polyfit(cdate, pop, 4)
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 5.429790e-20
```

```
p =
1.0e+05 *
0.0000   -0.0000   0.0000   -0.0126   6.0020
```
The warning arises because the `polyfit` function uses the `cdate` values as the basis for a matrix with very large values (it creates a Vandermonde matrix in its calculations – see the `polyfit` M-file for details). The spread of the `cdate` values results in scaling problems. One way to deal with this is to normalize the `cdate` data.

**Pre-Processing: Normalizing the Data**

Normalization is a process of scaling the numbers in a data set to improve the accuracy of the subsequent numeric computations. A way to normalize `cdate` is to scale it for zero mean and unit standard deviation:

\[
\text{sdate} = \frac{\text{cdate} - \text{mean(cdate)}}{\text{std(cdate)}}
\]

Now try the fourth-order polynomial model using the normalized data:

\[
p = \text{polyfit(sdate, pop, 4)}
\]

\[
p = 0.7047 \quad 0.9210 \quad 23.4706 \quad 73.8598 \quad 62.2285
\]

Evaluate the fitted polynomial at the normalized year values, and plot the fit against the observed data points:

\[
\text{pop4} = \text{polyval(p, sdate)};
\]

\[
\text{plot(cdate, pop4, '–', cdate, pop, '+'), grid on}
\]
Another way to normalize data is to use some knowledge of the solution and units. For example, with this data set, choosing 1790 to be year zero would also have produced satisfactory results.

**Analyzing Residuals**

A measure of the “goodness” of fit is the residual, the difference between the observed and predicted data. Compare the residuals for the various fits, using normalized date values:
### Case Study: Curve Fitting

**Fit**

```matlab
p1 = polyfit(sdate, pop, 1);
pop1 = polyval(p1, sdate);
plot(cdate, pop1, '-', cdate, pop, '+')
```

**Residuals**

```matlab
res1 = pop - pop1;
figure, plot(cdate, res1, '+')
```

**Linear fit appears unsatisfactory**
- note negative population values at lower end of scale.

**Fit**

```matlab
p = polyfit(sdate, pop, 2);
pop2 = polyval(p, sdate);
plot(cdate, pop2, '-', cdate, pop, '+')
```

**Residuals**

```matlab
res2 = pop - pop2;
figure, plot(cdate, res2, '+')
```

**Quadratic polynomial provides better fit to data points.**

**Residuals still appear strongly patterned.**
It's evident from studying the fit plots and residuals that it may be possible to do better than a simple polynomial fit with this data set.

### Exponential Fit

By looking at the population data plots on the previous pages, the population data curve is somewhat exponential in appearance. To take advantage of this,
let's try to fit the logarithm of the population values, again working with normalized year values.

```matlab
logp1 = polyfit(sdate, log10(pop), 1);
logpred1 = 10. ^ polyval(logp1, sdate);
semilogy(cdate, logpred1, '-', cdate, pop, '+');
grid on
```
Now try the logarithm analysis with a second-order model:

\[
\text{logp2} = \text{polyfit}(\text{sdate}, \text{log10(pop)}, 2);
\]

\[
\text{logpred2} = 10. \times \text{polyval} (\text{logp2}, \text{sdate});
\]

\[
\text{semilogy(cdate, logpred2, '–', cdate, pop, '+'); grid on}
\]

This is a more accurate model. The upper end of the plot appears to taper off, while the polynomial fits in the previous section continue, concave up, to infinity.
Now, if we view the residuals for the second-order logarithmic model:

\[
\text{logres2} = \log_{10}(\text{pop}) - \text{polyval(logp2, sdate)}; \\
\text{plot(cdate, logres2, ' + ')} \\
\]

\[
r = \text{pop} - 10^{(\text{polyval(logp2, sdate)})}; \\
\text{plot(cdate, r, ' + ')} \\
\]

The residuals are more random than for the simple polynomial fit. As might be expected, the residuals tend to get larger in magnitude as the population increases. But overall, the logarithmic model provides a more accurate fit to the population data.

**Error Bounds**

Error bounds are useful for determining if your data is reasonably modeled by the fit. An optional second output parameter can be obtained from `polyfit` and passed as an input parameter to `polyval` in order to obtain the error bounds.
For example, the code below uses `polyfit` and `polyval` to produce error bounds for a second-order polynomial model. Year values are normalized. This code uses an interval of $\pm 2\Delta$, corresponding to a 95% confidence interval.

```plaintext
[p2, S2] = polyfit(sdate, pop, 2);
[pop2, del2] = polyval(p2, sdate, S2);
plot(cdate, pop, '+', cdate, pop2, 'g-', cdate, pop2+2*del2, 'r:',
     cdate, pop2-2*del2, 'r:'), grid on
```

![Graph showing population growth over time with error bounds.](image)
Difference Equations and Filtering

MATLAB has functions for working with difference equations and filters. These functions operate primarily on vectors.

Vectors are used to hold sampled-data signals, or sequences, for signal processing and data analysis. For multi-input systems, each row of a matrix corresponds to a sample point with each input appearing as columns of the matrix.

The function

\[ y = \text{filter}(b, a, x) \]

processes the data in vector \( x \) with the filter described by vectors \( a \) and \( b \), creating filtered data \( y \).

The \text{filter} command can be thought of as an efficient implementation of the difference equation. The filter structure is the general tapped delay-line filter described by the difference equation below, where \( n \) is the index of the current sample, \( na \) is the order of the polynomial described by vector \( a \) and \( nb \) is the order of the polynomial described by vector \( b \). The output \( y(n) \), is a linear combination of current and previous inputs, \( x(n) \) \( x(n-1) \) ..., and previous outputs, \( y(n-1) y(n-2) \) ...

\[
a(1)y(n) = b(1)x(n) + b(2)x(n - 1) + \ldots + b(nb)x(n - nb + 1) \\
- a(2)y(n - 1) - \ldots - a(na)y(n - na + 1)
\]

Suppose, for example, we want to smooth our traffic count data with a moving average filter to see the average traffic flow over a four hour window. This process is represented by the difference equation

\[
y(n) = \frac{1}{4}x(n) + \frac{1}{4}x(n - 1) + \frac{1}{4}x(n - 2) + \frac{1}{4}x(n - 3)
\]

The corresponding vectors are

\[
a = 1;
\]
\[
b = [1/4 \ 1/4 \ 1/4 \ 1/4];
\]
Executing the command

\[
\text{load count.dat}
\]

creates the matrix \text{count} in the workspace.

For this example, extract the first column of traffic counts and assign it to the
vector \text{x},

\[
\text{x = count(:,1);}
\]

The four hour moving-average of the data is efficiently calculated with

\[
\text{y = filter(b, a, x);}
\]

Compare the original data and the smoothed data with an overlaid plot of the
two curves.

\[
\text{t = 1:length(x);}
\]
\[
\text{plot(t, x,'–.'}, t, y,'–'), grid on}
\]
\[
\text{legend('Original Data','Smoothed Data',2)
}\]

The filtered data represented by the solid line is the four-hour moving average
of the observed traffic count data represented by the dashed line.

For practical filtering applications, the Signal Processing Toolbox includes
numerous functions for designing and analyzing filters.
Fourier Analysis and the Fast Fourier Transform (FFT)

Fourier analysis is extremely useful for data analysis, as it breaks down a signal into constituent sinusoids of different frequencies. For sampled vector data, Fourier analysis is performed using the discrete Fourier transform (DFT).

The fast Fourier transform (FFT) is an efficient algorithm for computing the DFT of a sequence; it is not a separate transform. It is particularly useful in areas such as signal and image processing, where its uses range from filtering, convolution, and frequency analysis to power spectrum estimation.

MATLAB provides a collection of functions for computing and working with Fourier transforms:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fft</code></td>
<td>Discrete Fourier transform.</td>
</tr>
<tr>
<td><code>_fft2</code></td>
<td>Two-dimensional discrete Fourier transform.</td>
</tr>
<tr>
<td><code>fftn</code></td>
<td>N-dimensional discrete Fourier transform.</td>
</tr>
<tr>
<td><code>ifft</code></td>
<td>Inverse discrete Fourier transform.</td>
</tr>
<tr>
<td><code>ifft2</code></td>
<td>Two-dimensional inverse discrete Fourier transform.</td>
</tr>
<tr>
<td><code>ifftn</code></td>
<td>N-dimensional inverse discrete Fourier transform.</td>
</tr>
<tr>
<td><code>abs</code></td>
<td>Magnitude.</td>
</tr>
<tr>
<td><code>angle</code></td>
<td>Phase angle.</td>
</tr>
<tr>
<td><code>unwrap</code></td>
<td>Unwrap phase angle in radians.</td>
</tr>
<tr>
<td><code>fftshift</code></td>
<td>Move zeroth lag to center of spectrum.</td>
</tr>
<tr>
<td><code>cplxpair</code></td>
<td>Sort numbers into complex conjugate pairs.</td>
</tr>
<tr>
<td><code>nextpow2</code></td>
<td>Next higher power of two.</td>
</tr>
</tbody>
</table>

For length N input sequence x, the DFT is a length N vector, X. `fft` and `ifft` implement the relationships:
Data Analysis and Statistics

NOTE As the first element of a MATLAB vector has an index 1, the summations in the equations above are from 1 to $N$, producing identical results as the traditional Fourier equations with summations from 0 to $N-1$.

It is often more intuitive to represent the signal as the summation of sine and cosine functions, rather than in terms of complex exponentials. The relationship between the DFT and the Fourier coefficients $a$, $b$ is

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} X(k) e^{j2\pi (k-1)(n-1)/N} \quad 1 \leq n \leq N$$

$$X(k) = \sum_{n=1}^{N} x(n) e^{-j2\pi (k-1)(n-1)/N} \quad 1 \leq k \leq N$$

NOTE As the first element of a MATLAB vector has an index 1, the summations in the equations above are from 1 to $N$, producing identical results as the traditional Fourier equations with summations from 0 to $N-1$.

It is often more intuitive to represent the signal as the summation of sine and cosine functions, rather than in terms of complex exponentials. The relationship between the DFT and the Fourier coefficients $a$, $b$ is

$$x(n) = a_0 + \sum_{k=1}^{N/2} a(k) \cos \left( \frac{2\pi k t(n)}{N\Delta} \right) + b(k) \sin \left( \frac{2\pi k t(n)}{N\Delta} \right)$$

is

$$a_0 = 2X(1)/N$$

$$a(k) = 2 \cdot \text{real}(X(k+1))/N$$

$$b(k) = 2 \cdot \text{imag}(X(k+1))/N$$

where $x$ is a length $N$ discrete signal sampled at times $t$ with spacing $\Delta$.

The FFT of a column vector $x$

$$x = [4 \ 3 \ 7 \ -9 \ 1 \ 0 \ 0 \ 0]'$$

is found with

$$y = \text{fft}(x)$$
which results in

\[
y = \\
6.0000 \\
11.4853 - 2.7574i \\
-2.0000 - 12.0000i \\
-5.4853 + 11.2426i \\
18.0000 \\
-5.4853 - 11.2426i \\
-2.0000 + 12.0000i \\
11.4853 + 2.7574i
\]

Notice that although the sequence \( x \) is real, \( y \) is complex. The first component of the transformed data is the constant contribution and the fifth element corresponds to the Nyquist frequency. The last three values of \( y \) correspond to negative frequencies and, for the real sequence \( x \), they are complex conjugates of three components in the first half of \( y \).

Suppose, we want to analyze the variations in sunspot activity over the last 300 years. You are probably aware that sunspot activity is cyclical, reaching a maximum about every 11 years. Let’s confirm that.

Astromomers have tabulated a quantity called the Wolfer number for almost 300 years. This quantity measures both number and size of sunspots.
Load and plot the sunspot data:

```matlab
load sunspot.dat
year = sunspot(:,1);
wolfer = sunspot(:,2);
plot(year, wolfer)
title('Sunspot Data')
```

Now take the FFT of the sunspot data

```matlab
Y = fft(wolfer);
```

The result of this transform is the complex vector, \( Y \). The magnitude of \( Y \) squared is called the power and a plot of power versus frequency is a
"periodogram." Remove the first component of \( Y \), which is simply the sum of the data, and plot the results.

\[
N = \text{length}(Y);
Y(1) = [];
\text{power} = \text{abs}(Y(1:N/2)).^2;
\text{nyquist} = 1/2;
\text{freq} = (1:N/2)/(N/2)*\text{nyquist};
\text{plot(freq, power), grid on}
\text{xlabel('cycles/year')}
\text{title('Periodogram')}
\]
The scale in cycles/year is somewhat inconvenient. Let's plot in years/cycle and estimate what one cycle is. For convenience, plot the power versus period (where \( \text{period} = 1./\text{freq} \)) from 0 to 40 years/cycle.

```matlab
period = 1./freq;
plot(period, power), axis([0 40 0 2e7]), grid on
ylabel('Power')
xlabel('Period(Years/Cycle)')
```

In order to determine the cycle more precisely,

```matlab
[mp index] = max(power);
period(index)
```

ans =
11.0769

**Magnitude and Phase of Transformed Data**

Important information about a transformed sequence includes its magnitude and phase. The MATLAB functions `abs` and `angle` calculate this information.
To try this, create a time vector \( t \), and use this vector to create a sequence \( x \) consisting of two sinusoids at different frequencies:

\[
\begin{align*}
t &= 0:1/99:1; \\
x &= \sin(2\pi*15*t) + \sin(2\pi*40*t);
\end{align*}
\]

Now use the \texttt{fft} function to compute the DFT of the sequence. The code below calculates the magnitude and phase of the transformed sequence. It uses the \texttt{abs} function to obtain the magnitude of the data, the \texttt{angle} function to obtain the phase information, and \texttt{unwrap} to remove phase jumps greater than \( \pi \) to their \( 2\pi \) complement.

\[
\begin{align*}
y &= \text{fft}(x); \\
m &= \text{abs}(y); \\
p &= \text{unwrap} (\text{angle}(y));
\end{align*}
\]

Now create a frequency vector for the x-axis and plot the magnitude and phase:

\[
\begin{align*}
f &= (0:1\text{ length}(y) -1)^*99/\text{ length}(y); \\
\text{subplot}(2,1,1), \text{plot}(f,m), \\
\text{ylabel} (\text{'Abs. Magnitude'}, \text{grid on}) \\
\text{subplot}(2,1,2), \text{plot}(f,p*180/\pi) \\
\text{ylabel}(\text{'Phase [Degrees']}, \text{grid on}) \\
\text{xlabel}(\text{'Frequency [Hertz']})
\end{align*}
\]
The magnitude plot is perfectly symmetrical about the Nyquist frequency of 50 Hertz. The useful information in the signal is found in the range 0 to 50 Hertz.

**FFT Length Versus Speed**

You can add a second argument to `fft` to specify a number of points `n` for the transform:

\[ y = \text{fft}(x, n) \]

With this syntax, `fft` pads `x` with zeros if it is shorter than `n`, or truncates it if it is longer than `n`. If you do not specify `n`, `fft` defaults to the length of the input sequence.

The execution time for `fft` depends on the length of the transform:

- For any `n` that is a power of two, `fft` uses the high-speed radix-2 algorithm. This results in the fastest execution time. Additionally, the algorithm for power of two `n` is highly optimized for real `x`, providing a 40% increase in speed over the complex case.
- For any composite number `n` that is not a power of two, `fft` uses a prime factor algorithm. The speed of this algorithm depends on both the size of `n` and the number of prime factors it has. Although 1013 and 1000 are close in magnitude, `fft` transforms a sequence of length 1000 much more quickly than a sequence of length 1013.
- For a prime number `n`, `fft` cannot use an FFT algorithm. It instead performs the slower, computation-intensive DFT directly.

The inverse FFT function `ifft` also accepts a transform length argument.

For practical application of the FFT, the Signal Processing Toolbox includes numerous functions for spectral analysis.
# Function Functions

**Representing Functions in MATLAB** 7-3

**Plotting Mathematical Functions** 7-4

**Minimizing Functions and Finding Zeros** 7-7
- Minimizing Functions of One Variable 7-7
- Minimizing Functions of Several Variables 7-8
- Setting Minimization Options 7-9
- Finding Zeros of Functions 7-10
- Practicalities 7-12

**Numerical Integration (Quadrature)** 7-14
- Example: Computing the Length of a Curve 7-14
- Example: Double Integration 7-15
This chapter describes function functions, MATLAB functions that work with mathematical functions instead of numeric arrays. These function functions include:

- Numerical integration
- Optimization and nonlinear equation solution
- Differential equation solution

The function functions are located in the `funfun` directory in the MATLAB Toolbox.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization and root finding</td>
<td><code>fmin</code></td>
<td>Minimize function of one variable.</td>
</tr>
<tr>
<td></td>
<td><code>fmins</code></td>
<td>Minimize function of several variables.</td>
</tr>
<tr>
<td></td>
<td><code>fzero</code></td>
<td>Find zero of function of one variable.</td>
</tr>
<tr>
<td>Numerical integration (quadrature)</td>
<td><code>quad</code></td>
<td>Numerically evaluate integral, low order method.</td>
</tr>
<tr>
<td></td>
<td><code>quad8</code></td>
<td>Numerically evaluate integral, higher order method.</td>
</tr>
<tr>
<td></td>
<td><code>dblquad</code></td>
<td>Numerically evaluate double integral.</td>
</tr>
<tr>
<td>Plotting</td>
<td><code>ezplot</code></td>
<td>Easy to use function plotter.</td>
</tr>
<tr>
<td></td>
<td><code>fplot</code></td>
<td>Plot function.</td>
</tr>
</tbody>
</table>

The ordinary differential equation solvers are covered in Chapter 8.
Representing Functions in MATLAB

MATLAB represents mathematical functions by expressing them in M-files. For example, consider the function:

\[ f(x) = \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.9)^2 + 0.04} - 6 \]

This function can be used as input to any of the function functions. You can find it in the M-file named `humps.m`:

```matlab
function y = humps(x)
    y = 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;
```

All of the functions described in this chapter are called function functions because they accept, as one of their arguments, the name of an M-file like `humps` that defines a mathematical function.
Plotting Mathematical Functions

The `fplot` function plots a mathematical function between a given set of axes limits. You can control the x-axis limits only, or both the x- and y-axis limits. For example, to plot the `humps` function over the x-axis range \([-5 \ 5]\), use

```matlab
fplot('humps', [-5 5])
grid on
```
You can zoom in on the function, by selecting y-axis limits of -10 and 25, using

```
fplot('humps', [-5 5 -10 25])
grid on
```

You can also pass an expression for `fplot` to graph as in

```
fplot('2*sin(x+3)', [-1 1])
```

You can plot more than one function on the same graph with one call to `fplot`. If you use this with a M-file function, then the M-file must take a column vector \( x \) and return a matrix where each column corresponds to each function, evaluated at each value of \( x \).

If you pass an expression of several functions to `fplot`, it also must return a matrix where each column corresponds to each function evaluated at each value of \( x \), as in

```
fplot(['2*sin(x+3), humps(x)', [-1 1])
```
which plots the first and second expressions on the same graph.

Note that the expression

\[
\begin{bmatrix}
2 \sin(x+3), & \text{humps}(x)
\end{bmatrix}
\]

evaluates to a matrix of two columns, one for each function, when \(x\) is a column vector.
Minimizing Functions and Finding Zeros

MATLAB provides a number of high-level function functions that perform optimization-related tasks. This section describes:

- Minimizing a function of one variable.
- Minimizing a function of several variables.
- Setting minimization options.
- Finding a zero of a function of one variable.

For more sophisticated optimization capabilities, see the Optimization Toolbox.

Minimizing Functions of One Variable

Given a mathematical function of a single variable coded in an M-file, you can use the \texttt{fmin} function to find a local minimizer of the function in a given interval. For example, to find a minimum of the \texttt{humps} function in the range \([0.3, 1]\), use

\[
x = \text{fmin}('humps', 0.3, 1)
\]

which returns

\[
x = 0.6370
\]

You can ask for a tabular display of output by passing a fourth argument to \texttt{fmin} that has a nonzero value for its first element

\[
x = \text{fmin}('humps', 0.3, 1, 1)
\]
which gives the output

<table>
<thead>
<tr>
<th>Func evals</th>
<th>x</th>
<th>f(x)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.567376</td>
<td>12.9098</td>
<td>initial</td>
</tr>
<tr>
<td>2</td>
<td>0.732624</td>
<td>13.7746</td>
<td>golden</td>
</tr>
<tr>
<td>3</td>
<td>0.465248</td>
<td>25.1714</td>
<td>golden</td>
</tr>
<tr>
<td>4</td>
<td>0.644416</td>
<td>11.2693</td>
<td>parabolic</td>
</tr>
<tr>
<td>5</td>
<td>0.6413</td>
<td>11.2583</td>
<td>parabolic</td>
</tr>
<tr>
<td>6</td>
<td>0.637618</td>
<td>11.2529</td>
<td>parabolic</td>
</tr>
<tr>
<td>7</td>
<td>0.636985</td>
<td>11.2528</td>
<td>parabolic</td>
</tr>
<tr>
<td>8</td>
<td>0.637019</td>
<td>11.2528</td>
<td>parabolic</td>
</tr>
<tr>
<td>9</td>
<td>0.637052</td>
<td>11.2528</td>
<td>parabolic</td>
</tr>
</tbody>
</table>

\[ x = 0.6370 \]

This shows the current \( x \) and function value at \( x \) each time a function evaluation occurs. For \( \text{fmin} \), one function evaluation corresponds to one iteration of the algorithm. The last column shows what procedure is being used at each iteration, either a golden section search or a parabolic interpolation.

**Minimizing Functions of Several Variables**

The \( \text{fmins} \) function is similar to \( \text{fmin} \) except that it handles functions of many variables, and you specify a starting vector \( x_0 \) rather than a starting interval. \( \text{fmins} \) attempts to return a vector \( x \) that is a local minimizer of the mathematical function near this starting vector.

To try \( \text{fmins} \), create an M-file \text{three_var.m} that defines a function of three variables \( x, y, \) and \( z \):

```matlab
function b = three_var(v);
x = v(1);
y = v(2);
z = v(3);
b = x.^2 + 2.5*sin(y) – z^2*x^2*y^2;
```
Now find a minimum for this function, using \( x = -0.6 \), \( y = -1.2 \), and \( z = 0.135 \) as the starting values:

\[
\begin{align*}
\mathbf{v} &= [-0.6 -1.2 0.135]; \\
a &= \text{fmins('three_var', \mathbf{v})}
\end{align*}
\]

\[
a = \\
0.0000 \quad -1.5708 \quad 0.1664
\]

**Setting Minimization Options**

Optionally, you can specify a vector of control options that sets some minimization parameters by calling \text{fmin} with the syntax

\[
x = \text{fmin('function', x1, x2, options)}
\]

or \text{fmins} with the syntax

\[
x = \text{fmins('function', x0, options)}
\]

\text{options} is a vector of length 18 that is used by some specialized Optimization Toolbox functions. To set the \text{options} to default values, use

\[
\text{options} = \text{foptions};
\]

and then change values as needed such as

\[
\text{options}(1) = 1;
\]

to generate output at each iteration.

\text{fmin} and \text{fmins} use only four of the vector elements:

- \text{options}(1) is a flag that determines if intermediate steps in the minimization appear on the screen. For nonzero values, intermediate steps are displayed; for zero (default), no intermediate solutions are displayed.
- \text{options}(2) is the termination tolerance for \( x \). Its default value is 1. e-4.
- \text{options}(3) is the termination tolerance for function value. The default value is 1. e-4. This parameter is used by \text{fmins} but not \text{fmin}.
- \text{options}(14) is the maximum number of function evaluations allowed. The default value is 500 for \text{fmin} and 200*\text{length}(x0) for \text{fmins}. 

The number of function evaluations is returned in `options(10)` when you provide `fmin` or `fmins` with a second output argument as in

```
[x, options] = fmin('humps', 0.3, 1);
```
or

```
[a, options] = fmins('three_var', v);
```

### Finding Zeros of Functions

The `fzero` function attempts to find a zero of one equation with one variable. This function can be called with either a one-element starting point or a two-element vector that designates a starting interval. If you give `fzero` a starting point $x_0$, `fzero` first searches for an interval around this point where the function changes sign. If the interval is found, then `fzero` returns a value near where the function changes sign. If no such interval is found, `fzero` returns `NaN`. Alternatively, if you know two points where the function value differs in sign, you can specify this starting interval using a two-element vector, and `fzero` is guaranteed to narrow down the interval and return a value near the sign change.

Use `fzero` to find a zero of the `humps` function near $-0.2$

```
a = fzero('humps', -0.2)
a =
-0.1316
```

For this starting point, `fzero` searches in the neighborhood of $-0.2$ until it finds a change of sign between $-0.10949$ and $-0.264$. This interval is then narrowed to $-0.1316$. You can verify that $-0.1316$ has a function value very close to zero

```
humps(a)
ans =
 8.8818e-16
```
Suppose you know two places where the function value of *humps* differs in sign such as *x* = 1 and *x* = −1

```matlab
humps(1)
ans =
  16

humps(−1)
ans =
  −5.1378
```

Then you can give `fzero` this interval to start with and `fzero` then returns a point near where the function changes sign. You can display information as `fzero` progresses as well with:

```matlab
options = foptions;
options(1) = 1;
a = fzero('humps', [−1 1], [], options)
```

<table>
<thead>
<tr>
<th>Func evals</th>
<th>x</th>
<th>f(x)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−1</td>
<td>−5.13779</td>
<td>initial</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>16</td>
<td>initial</td>
</tr>
<tr>
<td>2</td>
<td>−0.513876</td>
<td>−4.02235</td>
<td>interpolation</td>
</tr>
<tr>
<td>3</td>
<td>0.243062</td>
<td>71.6382</td>
<td>bisection</td>
</tr>
<tr>
<td>4</td>
<td>−0.473635</td>
<td>−3.83767</td>
<td>interpolation</td>
</tr>
<tr>
<td>5</td>
<td>−0.115287</td>
<td>0.414441</td>
<td>bisection</td>
</tr>
<tr>
<td>6</td>
<td>−0.150214</td>
<td>−0.423446</td>
<td>interpolation</td>
</tr>
<tr>
<td>7</td>
<td>−0.132562</td>
<td>−0.0226907</td>
<td>interpolation</td>
</tr>
<tr>
<td>8</td>
<td>−0.131666</td>
<td>−0.0011492</td>
<td>interpolation</td>
</tr>
<tr>
<td>9</td>
<td>−0.131618</td>
<td>1.88371e−07</td>
<td>interpolation</td>
</tr>
<tr>
<td>10</td>
<td>−0.131618</td>
<td>−2.7935e−11</td>
<td>interpolation</td>
</tr>
<tr>
<td>11</td>
<td>−0.131618</td>
<td>8.88178e−16</td>
<td>interpolation</td>
</tr>
<tr>
<td>12</td>
<td>−0.131618</td>
<td>−9.76996e−15</td>
<td>interpolation</td>
</tr>
</tbody>
</table>

```matlab
a =
  -0.1316
```
The steps of the algorithm include both bisection and interpolation under the Procedure column. If the example had started with a scalar starting point instead of an interval, the first steps after the initial function evaluations would have included some search steps while \texttt{fzero} searched for an interval containing a sign change.

Optionally, you can specify a relative error tolerance as a third input argument. In the call above, passing in the empty matrix causes the default relative error tolerance of \texttt{eps} to be used.

**Practicalities**
Optimization problems may take many iterations to converge. Most optimization problems benefit from good starting guesses. This improves the execution efficiency and may help locate the global minimum instead of a local minimum.

Sophisticated problems are best solved by an evolutionary approach whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

Below is a list of typical problems and recommendations for dealing with them:
## Troubleshooting

<table>
<thead>
<tr>
<th>Problem</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The solution found by <code>fmin</code> or <code>fmins</code> does not appear to be a global minimum.</td>
<td>There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points (or intervals in the case of <code>fmin</code>) may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.</td>
</tr>
<tr>
<td>Sometimes an optimization problem has values of $x$ for which it is impossible to evaluate $f$.</td>
<td>Modify your function to include a penalty function to give a large positive value to $f$ when infeasibility is encountered.</td>
</tr>
<tr>
<td>The minimization routine appears to enter an infinite loop or returns a solution that is not a minimum (or not a zero in the case of <code>fzero</code>).</td>
<td>Your objective function may be returning <code>Inf</code>, <code>NaN</code> or complex values. The optimization routines expect only real numbers to be returned. Any other values may cause unexpected results. Insert code into your objective function M-file to verify that only real numbers are returned (use the functions <code>isreal</code> and <code>isfinite</code>).</td>
</tr>
</tbody>
</table>
Numerical Integration (Quadrature)

The area beneath a section of a function $F(x)$ can be determined by numerically integrating $F(x)$, a process referred to as quadrature. The two MATLAB functions for one-dimensional quadrature are

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quad</td>
<td>Adaptive Simpson's rule</td>
</tr>
<tr>
<td>quad8</td>
<td>Adaptive Newton Cotes 8 panel rule</td>
</tr>
</tbody>
</table>

To integrate the function defined by `humps.m` from 0 to 1, use

```matlab
q = quad('humps', 0, 1)
```

$q = 29.8583$

Both `quad` and `quad8` operate recursively. If either method reaches the maximum number of 10 recursive calls, the method returns a value of `Inf` indicating possible singularity.

You can include a fourth argument for `quad` or `quad8` that specifies a relative error tolerance for the integration. If this fourth argument is a two-element vector, its first element specifies a relative tolerance and its second an absolute tolerance. If a nonzero fifth argument is passed to `quad` or `quad8`, the function evaluations are traced with a point plot of the integrand.

**Example: Computing the Length of a Curve**

You can use `quad` or `quad8` to compute the length of a curve. Consider the curve parameterized by the equations

$$
\begin{align*}
    x(t) &= \sin(2t), & y(t) &= \cos(t), & z(t) &= t
\end{align*}
$$

where $t \in [0, 3\pi]$.

A three-dimensional plot of this curve is

```matlab
t = 0:0.1:3*pi;
plot3(sin(2*t), cos(t), t)
```
The arc length formula says the length of the curve is the integral of the norm of the derivatives of the parameterized equations

\[ \int_{0}^{3\pi} \sqrt{4\cos(2t)^2 + \sin(t)^2 + 1} \ dt \]

The function `hcurve` computes the integrand

```matlab
function f = hcurve(t)
f = sqrt(4*cos(2*t).^2 + sin(t).^2 + 1);
end
```

Integrate this function with a call to `quad`

```matlab
len = quad('hcurve', 0, 3*pi)
len =
1.7222e+01
```

and so the length of this curve is about 17.2.

**Example: Double Integration**

Consider the numerical solution of

\[ \int_{y_{\text{min}}}^{y_{\text{max}}} \int_{x_{\text{min}}}^{x_{\text{max}}} f(x, y) \ dx \ dy \]

For this example, \( f(x, y) = y\sin(x) + x\cos(y) \). The first step is to build the function to be evaluated. The function must be capable of returning a vector output when given a vector input. You must also consider which variable is in the inner integral, and which goes in the outer integral. In this example, the inner variable is \( x \) and the outer variable is \( y \) (the order in the integral is \( dx \ dy \)).

In this case, the integrand function is:

```matlab
function out = integrnd(x, y)
out = y*sin(x) + x*cos(y);
end
```

To perform the integration, two functions are available in the `funfun` directory. The first, `dblquad`, is called directly from the command line. This M-file evaluates the outer loop using `quad`. At each iteration, `quad` calls the second helper function that evaluates the inner loop.
To evaluate the double integral, use

\[
\text{result} = \text{dblquad('integrnd', xmin, xmax, ymin, ymax)};
\]

The first argument is a string with the name of the integrand function; the second to fifth arguments are:

- \(xmin\): lower limit of inner integral
- \(xmax\): upper limit of the inner integral
- \(ymin\): lower limit of outer integral
- \(ymax\): upper limit of the outer integral

Here is a numeric example that illustrates the use of \text{dblquad}:

\[
\begin{align*}
xmin &= \pi; \\
xmax &= 2*\pi; \\
ymin &= 0; \\
ymax &= \pi; \\
\text{result} &= \text{dblquad('integrnd', xmin, xmax, ymin, ymax)}
\end{align*}
\]

The result is \(-9.8698\).

By default, \text{dblquad} calls \text{quad}. To integrate the previous example using \text{quad8} (with the default values for the tolerance and trace arguments), use

\[
\begin{align*}
\text{result} &= \text{dblquad('integrnd', xmin, xmax, ymin, ymax, [], 'quad8')}
\end{align*}
\]

Alternatively, any user-defined quadrature function name may be passed to \text{dblquad} as long as the quadrature function has the same calling and return arguments as \text{quad}.

\textbf{NOTE} For details on another set of function functions, ordinary differential equation solvers, see Chapter 8.
Ordinary Differential Equations

Quick Start ........................................ 8-3
Representing Problems ......................... 8-5
ODE Solvers ....................................... 8-10
Creating ODE Files .............................. 8-14
Improving Solver Performance ............... 8-17
Examples: Applying the ODE Solvers ...... 8-33
Questions and Answers ....................... 8-47
This chapter describes how to use MATLAB to solve initial value problems of ordinary differential equations (ODEs). It discusses how to represent initial value problems (IVPs) in MATLAB and how to apply MATLAB’s ODE solvers to such problems. It explains how to select a solver, and how to specify solver options for efficient, customized execution. The chapter also includes a trouble shooting guide and an extensive “Examples” section.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinary differential equation solvers</td>
<td>ode45</td>
<td>Nonstiff differential equations, medium order method.</td>
</tr>
<tr>
<td></td>
<td>ode23</td>
<td>Nonstiff differential equations, low order method.</td>
</tr>
<tr>
<td></td>
<td>ode113</td>
<td>Nonstiff differential equations, variable order method.</td>
</tr>
<tr>
<td></td>
<td>ode15s</td>
<td>Stiff differential equations, variable order method.</td>
</tr>
<tr>
<td></td>
<td>ode23s</td>
<td>Stiff differential equations, low order method.</td>
</tr>
<tr>
<td>ODE option handling</td>
<td>odeset</td>
<td>Create/alter ODE OPTIONS structure.</td>
</tr>
<tr>
<td></td>
<td>odeget</td>
<td>Get ODE OPTIONS parameters.</td>
</tr>
<tr>
<td>ODE output functions</td>
<td>odeplot</td>
<td>Time series plots.</td>
</tr>
<tr>
<td></td>
<td>odephas2</td>
<td>Two-dimensional phase plane plots.</td>
</tr>
<tr>
<td></td>
<td>odephas3</td>
<td>Three-dimensional phase plane plots.</td>
</tr>
<tr>
<td></td>
<td>odeprint</td>
<td>Print to command window.</td>
</tr>
</tbody>
</table>
Quick Start

1. Write the ordinary differential equation $y^{(n)} = f(t, y, y', ..., y^{(n-1)})$ as a system of first-order equations by making the substitutions

$$y_1 = y, y_2 = y', ..., y_n = y^{(n-1)}$$

Then:

$$
y_1' = y_2 \\
y_2' = y_3 \\
... \\
y_n' = f(t, y_1, y_2, ..., y_n)
$$

is a system of $n$ first-order ODEs. For example, consider the initial value problem

$$y''' - 3y'' - y' y = 0 \quad y(0) = 0 \quad y'(0) = 1 \quad y''(0) = -1$$

Solve the differential equation for its highest derivative, writing $y'''$ in terms of $t$ and its lower derivatives $y''' = 3y'' + y'y$. If you let $y_1 = y, y_2 = y'$, and $y_3 = y''$, then

$$
y_1' = y_2 \\
y_2' = y_3 \\
y_3' = 3y_3 + y_2y_1
$$

is a system of three first-order ODEs with initial conditions

$$y_1(0) = 0 \quad y_2(0) = 1 \quad y_3(0) = -1$$
Note that the IVP now has the form \( Y' = F(t, Y), Y(0) = Y_0 \), where \( Y = [y_1; y_2; y_3] \).

2 Code the first-order system in an M-file that accepts two arguments, \( t \) and \( y \), and returns a column vector:

```matlab
function dy = F(t,y)
    dy = [y(2); y(3); 3*y(3) + y(2)*y(1)];
```

This ODE file must accept the arguments \( t \) and \( y \), although it does not have to use them. Here, the vector \( dy \) must be a column vector.

3 Apply a solver function to the problem. The general calling syntax for the ODE solvers is

\[
[T, Y] = \text{solver}('F', tspan, y0)
\]

where \( \text{solver} \) is a solver function like \texttt{ode45}. The input arguments are:

- \( F \): String containing the ODE file name
- \( tspan \): Vector of time values where \([t_0 \ t_{\text{final}}]\) causes the solver to integrate from \( t_0 \) to \( t_{\text{final}} \)
- \( y0 \): Column vector of initial conditions at the initial time \( t_0 \)

For example, to use the \texttt{ode45} solver to find a solution of the sample IVP on the time interval \([0 \ 1]\), the calling sequence is

\[
[T, Y] = \text{ode45}('F', [0 \ 1], [0; 1; -1])
\]

Each row in solution array \( Y \) corresponds to a time returned in column vector \( T \). Also, in the case of the sample IVP, \( Y(:,1) \) is the solution, \( Y(:,2) \) is the derivative of the solution, and \( Y(:,3) \) is the second derivative of the solution.
Representing Problems

This section describes how to represent ordinary differential equations as systems for the MATLAB ODE solvers.

The MATLAB ODE solvers are designed to handle ordinary differential equations. These are differential equations containing one or more derivatives of a dependent variable \( y \) with respect to a single independent variable \( t \), usually referred to as time. The derivative of \( y \) with respect to \( t \) is denoted as \( y' \), the second derivative as \( y'' \), and so on. Often \( y(t) \) is a vector, having elements \( y_1, y_2, \ldots, y_n \).

ODEs often involve a number of dependent variables, as well as derivatives of order higher than one. To use the MATLAB ODE solvers, you must rewrite such equations as an equivalent system of first-order differential equations in terms of a vector \( y \) and its first derivative:

\[
y' = F(t, y)
\]

Once you represent the equation in this way, you can code it as an ODE M-file that a MATLAB ODE solver can use.

**Initial Value Problems and Initial Conditions**

Generally there are many functions \( y(t) \) that satisfy a given ODE, and additional information is necessary to specify the solution of interest. In an initial value problem, the solution of interest has a specific initial condition, that is, \( y \) is equal to \( y_0 \) at a given initial time \( t_0 \). An initial value problem for an ODE is then

\[
y' = F(t, y) \\
y(t_0) = y_0
\]

If the function \( F(t, y) \) is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate \( y(t) \) by numerical means, such as one of the solvers of the MATLAB ODE suite.
**Example: The van der Pol Equation**

An example of an ODE is the van der Pol equation

\[
y_1'' - \mu (1 - y_1^2) y_1' + y_1 = 0
\]

where \( \mu > 0 \) is a scalar parameter.

**Rewriting the System**

To express this equation as a system of first-order differential equations for MATLAB, introduce a variable \( y_2 \) such that \( y_1' = y_2 \). You can then express this system as

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= \mu (1 - y_1^2) y_2 - y_1
\end{align*}
\]

**Writing the ODE File**

The code below shows how to represent the van der Pol system in a MATLAB ODE file, an M-file that describes the system to be solved. An ODE file always accepts at least two arguments, \( t \) and \( y \). This simple two line file assumes a value of 1 for \( \mu \). \( y_1 \) and \( y_2 \) become \( y(1) \) and \( y(2) \), elements in a two-element vector.

```matlab
function dy = vdp1(t,y)
    dy = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

**NOTE** This ODE file does not actually use the \( t \) argument in its computations. It is not necessary for it to use the \( y \) argument either – in some cases, for example, it may just return a constant. The \( t \) and \( y \) variables, however, must always appear in the input argument list.

**Calling the Solver**

Once the ODE system is coded in an ODE file, you can use the MATLAB ODE solvers to solve the system on a given time interval with a particular initial condition vector. For example, to use `ode45` to solve the van der Pol equation...
on time interval [0 20] with an initial value of 2 for y(1) and an initial value of 0 for y(2):

\[
[T, Y] = \text{ode45('vdp1', [0 20], [2; 0])};
\]

The resulting output [T,Y] is a column vector of time points T and a solution array Y. Each row in solution array Y corresponds to a time returned in column vector T.

**Viewing the Results**

Use the \texttt{plot} command to view solver output:

\begin{verbatim}
plot(t, y(:,1), '-', t, y(:,2), '--');
title('Solution of van der Pol Equation, \(\mu = 1\)');
xlabel('time \(t\)');
ylabel('solution \(y\)');
legend('y1', 'y2')
\end{verbatim}
Example: The van der Pol Equation, $\mu = 1000$ (Stiff)

Stiff ODE Problems

This section presents a stiff problem. For a stiff problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change.

When $\mu$ is increased to 1000, the solution to the van der Pol equation changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes a more difficult task. Because this particular problem is stiff, a nonstiff solver such as ode45 is so inefficient that it is impractical. The stiff solver ode15s is intended for such problems.

This code shows how to represent the van der Pol system in an ODE file with $\mu = 1000$.

```matlab
function dy = vdp1000(t,y)
    dy = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
end
```

Now use the ode15s function to solve vdp1000. Retain the initial condition vector of $[2; 0]$, but use a time interval of $[0 3000]$. For scaling purposes, plot just the first component of $y(t)$:

```matlab
[t,y] = ode15s('vdp1000',[0 3000],[2; 0]);
plot(t,y(:,1),'o');
title('Solution of van der Pol Equation, $\mu = 1000$');
xlabel('time $t$');
ylabel('solution $y(:,1)$');
```
Solution of van der Pol Equation, \( \mu = 1000 \)
OE Solvers

The MATLAB ODE solver functions implement numerical integration methods. Beginning at the initial time and with initial conditions, they step through the time interval, computing a solution at each time step. If the solution for a time step satisfies the solver's error tolerance criteria, it is a successful step. Otherwise, it is a failed attempt; the solver shrinks the step size and tries again.

This section describes how to represent problems for use with the MATLAB solvers and how to optimize solver performance. You can also use the online help facility to get information on the syntax for any function, as well as information on demo files for these solvers.

Nonstiff Solvers

There are three solvers designed for nonstiff problems:

- **ode45** is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver – in computing \( y(t_n) \), it needs only the solution at the immediately preceding time point, \( y(t_{n-1}) \). In general, **ode45** is the best function to apply as a “first try” for most problems.
- **ode23** is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than **ode45** at crude tolerances and in the presence of mild stiffness. Like **ode45**, **ode23** is a one-step solver.
- **ode113** is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than **ode45** at stringent tolerances and when the ODE function is particularly expensive to evaluate. **ode113** is a multistep solver – it normally needs the solutions at several preceding time points to compute the current solution.

Stiff Solvers

Not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Stiff solvers can be used exactly like the other solvers. However, you can often significantly improve the efficiency of the stiff solvers by providing them with additional information about the problem. See “Improving Solver Performance” on page 8-17 for details on how to provide this information, and for details on how to change solver parameters such as error tolerances.
There are two solvers designed for stiff problems:

- **ode15s** is a variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs, (also known as Gear’s method) that are usually less efficient. Like ode113, ode15s is a multistep solver. If you suspect that a problem is stiff or if ode45 failed or was very inefficient, try ode15s.

- **ode23s** is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective.

**ODE Solver Basic Syntax**

All of the ODE solver functions share a syntax that makes it easy to try any of the different numerical methods if it is not apparent which is the most appropriate. To apply a different method to the same problem, simply change the ODE solver function name. The simplest syntax, common to all the solver functions, is

```matlab
[T, Y] = solver('F', tspan, y0)
```

where **solver** is one of the ODE solver functions listed previously. The input arguments are:

- **'F'**  
  String containing the name of the file that describes the system of ODEs.

- **tspan**  
  Vector specifying the interval of integration. For a two-element vector `tspan = [t0 tfinal]`, the solver integrates from `t0` to `tfinal`. For `tspan` vectors with more than two elements, the solver returns solutions at the given time points, as described below. Note that `t0 > tfinal` is allowed.

- **y0**  
  Vector of initial conditions for the problem.
The output arguments are:

\[
\begin{align*}
T & \quad \text{Column vector of time points} \\
Y & \quad \text{Solution array. Each row in Y corresponds to the solution at a time returned in the corresponding row of T.}
\end{align*}
\]

**Obtaining Solutions at Specific Time Points**

To obtain solutions at specific time points \(t_0, t_1, \ldots, t_{\text{final}}\), specify \(t\text{span}\) as a vector of the desired times. The time values must be in order, either increasing or decreasing.

Specifying these time points in the \(t\text{span}\) vector does not affect the internal time steps that the solver uses to traverse the interval from \(t\text{span}(1)\) to \(t\text{span}(\text{end})\) and has little effect on the efficiency of computation. All solvers in the MATLAB ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in \(t\text{span}\), the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points.

**Specifying Solver Options**

In addition to the simple syntax, all of the ODE solvers accept a fourth input argument, \(\text{options}\), which can be used to change the default integration parameters.

\[
[t, y] = \text{solver('F', t\text{span}, y0, options)}
\]

The \(\text{options}\) argument is created with the \text{odeset} function (see “Creating an Options Structure: The \text{odeset} Function” on page 8-20). Any input parameters after the \(\text{options}\) argument are passed to the ODE file every time it is called. For example,

\[
[T, Y] = \text{solver('F', t\text{span}, y0, options, p1, p2, \ldots)}
\]

calls

\[F(t, y, \text{flag}, p1, p2, \ldots)\]
Obtaining Statistics About Solver Performance
Use an additional output argument $S$ to obtain statistics about the ODE solver’s computations:

$$ [T,Y,S] = \text{solver}('F',t\text{span},y0,\text{options},...) $$

$S$ is a six-element column vector:

- Element 1 is the number of successful steps.
- Element 2 is the number of failed attempts.
- Element 3 is the number of times the ODE file was called to evaluate $F(t,y)$.
- Element 4 is the number of times that the partial derivatives matrix $\frac{\partial F}{\partial y}$ was formed.
- Element 5 is the number of LU decompositions.
- Element 6 is the number of solutions of linear systems.

The last three elements of the list apply to the stiff solvers only.

The solver automatically displays these statistics if the $\text{Stats}$ property (see page page 8-25) is set in the $\text{options}$ argument.
Creating ODE Files

The van der Pol examples in the previous sections show some simple ODE files. This section provides more detail and describes how to create more advanced ODE files that can accept additional input parameters and return additional information.

ODE File Overview

Look at the simple ODE file vdp1.m from earlier in this chapter:

```matlab
function dy = vdp1(t, y)
    dy = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Although this is a simple example, it demonstrates two important requirements for ODE files:

- The first two arguments must be \( t \) and \( y \).
- By default, the ODE file must return a column vector \( F(t, y) \).

Defining the Initial Values in the ODE File

It is possible to specify default \( tspan \), \( y0 \) and \( options \) in the ODE file, defining the entire initial value problem in the one file. In this case, the solver can be called as

```matlab
[T, Y] = solver('F', [], []);
```

The solver extracts the default values from the ODE file. You can also omit empty arguments at the end of the argument list. For example,

```matlab
[T, Y] = solver('F');
```

When you call a solver with an empty or missing \( tspan \) or \( y0 \), the solver calls the specified ODE file to obtain any values not supplied in the solver argument list. It uses the syntax

```matlab
[tspan, y0, options] = F([], [], 'init')
```
The ODE file is then expected to return three outputs:

- Output 1 is the \( tspan \) vector.
- Output 2 is the initial value, \( y_0 \).
- Output 3 is either an options structure created with the `odeset` function or an empty matrix \([\ ]\).

### Coding the ODE File to Return Initial Values

If you use this approach, your ODE file must check the value of the third argument and return the appropriate output. For example, you can modify the van der Pol ODE file `vdp1.m` to check the third argument, `flag`, and return either the default vector \( F(t, y) \) or \([tspan, y0, options]\) depending on the value of `flag`:

```matlab
function [out1, out2, out3] = vdp1(t, y, flag)
    if nargin < 3 | isempty(flag)
        % Return \( dy/dt = F(t, y) \).
        out1 = [y(2); (1-y(1)^2)*y(2)-y(1)];
    elseif strcmp(flag,'init')
        % Return \([tspan, y0, options]\).
        out1 = [0; 20]; % tspan
        out2 = [2; 0]; % initial conditions
        out3 = odeset('RelTol',1e-4); % options
    end

NOTE The third argument, referred to as the `flag` argument, is a special argument that notifies the ODE file that the solver is expecting a specific kind of information. The `'init'` string, for initial values, is just one possible value for this flag. For complete details on the `flag` argument, see “Special Purpose ODE Files and the flag Argument” on page 8-18.
Passing Additional Parameters to the ODE File

In some cases your ODE system may require additional parameters beyond the required \(t\) and \(y\) arguments. For example, you can generalize the van der Pol ODE file by passing it a \(\mu\) parameter, instead of specifying a value for \(\mu\) explicitly in the code:

```matlab
function [out1, out2, out3] = vdpode(t, y, flag, mu)
if nargin < 4 | isempty(mu)
    mu = 1;
end
if nargin < 3 | isempty(flag)
    % Return \(dy/dt = F(t,y)\).
    out1 = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
else if strcmp(flag,'init')
    % Return \([tspan,y0,options]\).
    out1 = [0; 20]; % tspan
    out2 = [2; 0]; % initial conditions
    out3 = odeset('RelTol',1e-4); % options
end
```

In this example, the parameter \(\mu\) is an optional argument specific to the van der Pol example. MATLAB and the ODE solvers do not set a limit on the number of parameters you can pass to an ODE file.

Guidelines for Creating ODE Files

- The \(ode\) file must have at least two input arguments, \(t\) and \(y\). It is not necessary, however, for the function to use either \(t\) or \(y\).
- The derivatives returned by \(F(t,y)\) must be a column vector.
- Any additional parameters beyond \(t\) and \(y\) must appear at the end of the argument list and must begin at the fourth input parameter. The third position is reserved for an optional flag, as shown above in “Coding the ODE File to Return Initial Values.” The \(flag\) argument is described in more detail in “Special Purpose ODE Files and the \(flag\) Argument” on page page 8-18.
In some cases, you can improve ODE solver performance by specially coding your ODE file. For instance, you might accelerate the solution of a stiff problem by coding the ODE file to compute the Jacobian matrix analytically.

Another way to improve solver performance, often used in conjunction with a specially coded ODE file, is to tune solver parameters. The default parameters in the ODE solvers are selected to handle common problems. In some cases, however, tuning the parameters for a specific problem can improve performance significantly. You do this by supplying the solvers with one or more property values contained within an options argument.

\[
[T, Y] = \\text{solver}('F', tspan, y0, \text{options})
\]

The property values within the options argument are created with the odeset function, in which named properties are given specified values.

<table>
<thead>
<tr>
<th>Category</th>
<th>Property Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error tolerance</td>
<td>RelTol, AbsTol</td>
<td>8-21</td>
</tr>
<tr>
<td>Solver output</td>
<td>OutPutFcn, OutPutSel, Refine, Stats</td>
<td>8-22</td>
</tr>
<tr>
<td>Jacobian matrix</td>
<td>Jacobian, JConstant, JPattern, Vectorized</td>
<td>8-25</td>
</tr>
<tr>
<td>Step size</td>
<td>InitialStep, MaxStep</td>
<td>8-28</td>
</tr>
<tr>
<td>Mass matrix</td>
<td>Mass, MassConstant</td>
<td>8-29</td>
</tr>
<tr>
<td>Event location</td>
<td>Events</td>
<td>8-30</td>
</tr>
<tr>
<td>ode15s</td>
<td>MaxOrder, BDF</td>
<td>8-32</td>
</tr>
</tbody>
</table>
Special Purpose ODE Files and the flag Argument

The MATLAB ODE solvers are capable of using additional information provided in the ODE file. In this more general use, an ODE file is expected to respond to the arguments `odefile(t, y, flag, p1, p2, . . .)` where `t` and `y` are the integration variables, `flag` is a string indicating the type of information that the ODE file should return, and `p1, p2, . . .` are any additional parameters that the problem requires. The currently supported flags are:

<table>
<thead>
<tr>
<th>Flags</th>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘’ (empty)</td>
<td><code>F(t, y)</code></td>
</tr>
<tr>
<td>‘init’</td>
<td><code>tspan, y0, and options</code> for this problem</td>
</tr>
<tr>
<td>‘jacobian’</td>
<td>Jacobian matrix <code>J (t, y) = ∂F/∂y</code></td>
</tr>
<tr>
<td>‘jpattern’</td>
<td>Matrix showing the Jacobian sparsity pattern</td>
</tr>
<tr>
<td>‘mass’</td>
<td>Mass matrix <code>M(t)</code> for solving <code>M(t)y' = F(t, y)</code></td>
</tr>
<tr>
<td>‘events’</td>
<td>Information to define an event location problem</td>
</tr>
</tbody>
</table>

The template below illustrates how to code an extended ODE file that uses the `switch` construct and the ODE file’s third input argument, `flag`, to supply additional information. For illustration, the file also accepts two additional input parameters `p1` and `p2`. 
NOTE: The example below is only a template. In your own coding you should not include all of the cases shown. For example, 'jacobian' information is used for evaluating Jacobians analytically, and 'jpattern' information is used for generating Jacobians numerically.

function [out1, out2, out3] = odefile(t, y, flag, p1, p2)
if nargin < 3 | isempty(flag)
    % Return dy/dt = f(t, y).
    out1 = < Insert a function of t and/or y, p1 and p2 here. >;
else
    switch(flag)
        case 'init' % Return [tspan, y0, options].
            out1 = < Insert tspan here. >;
            out2 = < Insert y0 here. >;
            out3 = < Insert options = odeset(...) or [] here. >;
        case 'jacobian' % Return matrix J(t,y) = df/dy.
            out1 = < Insert or evaluate Jacobian matrix here. >;
        case 'jpattern' % Return sparsity pattern matrix S.
            out1 = < Insert Jacobian matrix sparsity pattern here. >;
        case 'mass' % Return mass matrix M(t) or M
            out1 = < Insert or evaluate mass matrix here. >;
        case 'events' % Return information for event location.
            out1 = < Insert vector of event functions here. >;
            out2 = < Insert logical isteminal vector here. >;
            out3 = < Insert direction vector here. >;
        otherwise
            error(['Unknown flag ' flag '.']);
    end
end
Creating an Options Structure: The odeset Function

The `odeset` function creates an options structure that you can supply to any of the ODE solvers. `odeset` accepts property name/property value pairs using the syntax

\[ \text{options} = \text{odeset('name1', value1, 'name2', value2, \ldots)} \]

This creates a structure `options` in which the named properties have the specified values. Any unspecified properties contain default values in the solvers. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. `odeset` ignores case for property names.

With no input arguments, `odeset` displays all property names and their possible values, indicating defaults with `{ }`:

- `AbsTol`: [ positive scalar or vector {1e–6} ]
- `BDF`: [ on | {off} ]
- `Events`: [ on | {off} ]
- `InitialStep`: [ positive scalar ]
- `Jacobian`: [ on | {off} ]
- `JConstant`: [ on | {off} ]
- `JPattern`: [ on | {off} ]
- `Mass`: [ on | {off} ]
- `MassConstant`: [ on | {off} ]
- `MaxOrder`: [ 1 | 2 | 3 | 4 | {5} ]
- `MaxStep`: [ positive scalar ]
- `OutputFcn`: [ string ]
- `OutputSel`: [ vector of integers ]
- `Refine`: [ positive integer ]
- `RelTol`: [ positive scalar {1e–3} ]
- `Stats`: [ on | {off} ]
- `Vectorized`: [ on | {off} ]

Modifying an Existing Options Structure

To modify an existing options argument, use

\[ \text{options} = \text{odeset(oldopts, 'name1', value1, \ldots)} \]

This sets `options` equal to the existing structure `oldopts`, overwriting any values in `oldopts` that are respecified using name/value pairs and adding to
the structure any new pairs. The modified structure is returned as an output argument. In the same way, the command

```matlab
options = odeset(oldopts, newopts)
```

combines the structures of `oldopts` and `newopts`. In the output argument, any values in the second argument (other than the empty matrix) overwrite those in the first argument.

**Querying Options: The `odeget` Function**

The solvers use the `odeget` function to extract property values from an `options` structure created with `odeset`:

```matlab
o = odeget(options, 'name')
```

This returns the value of the specified property, or an empty matrix `[]` if the property value is unspecified in the `options` structure.

As with `odeset`, it is sufficient to type only the leading characters that uniquely identify the property name, and case is ignored for property names.

**Error Tolerance Properties**

The solvers use standard local error control techniques for monitoring and controlling the error of each integration step. At each step, the local error `e` in the `i`'th component of the solution is estimated and is required to be less than or equal to the acceptable error, which is a function of two user-defined tolerances `RelTol` and `AbsTol`:

\[ |e(i)| \leq \max(RelTol \times |y(i)|, AbsTol(i)) \]

- `RelTol` is the relative accuracy tolerance, a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in the answer. The default, `1e-3`, corresponds to 0.1% accuracy.
- `AbsTol` is a scalar or vector of the absolute error tolerances for each solution component. `AbsTol(i)` is a threshold below which the values of the corresponding solution components are unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. The default value is `1e-6`. 
Set tolerances using `odeset`, either at the command line or in the ODE file.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel Tol</td>
<td>Positive scalar</td>
<td>A relative error tolerance that applies to all components of the solution vector ( y ). Default value is ( 10^{-3} ) (0.1% accuracy).</td>
</tr>
<tr>
<td></td>
<td>{1e-3}</td>
<td></td>
</tr>
<tr>
<td>Abs Tol</td>
<td>Positive scalar</td>
<td>Absolute error tolerances that apply to the corresponding components of the solution vector. If a scalar value is specified, it applies to all components of the solution vector ( y ). Default value is ( 10^{-6} ).</td>
</tr>
<tr>
<td></td>
<td>or vector {1e-6}</td>
<td></td>
</tr>
</tbody>
</table>

The ODE solvers are designed to deliver, for routine problems, accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over “long” intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `Rel Tol`. For the absolute error tolerance, the scaling of the solution components is important: if \( |y| \) is somewhat smaller than `Abs Tol`, the solver is not constrained to obtain any correct digits in \( y \). You might have to solve a problem more than once to discover the scale of solution components.

**Solver Output Properties**

The solver output properties available with `odeset` let you control the output that the solvers generate. With these properties, you can specify an output function, a function that executes if you call the solver with no output arguments. In addition, the ODE solver output options let you obtain additional solutions at equally spaced points within each time step, or view statistics about the computations.
OutputFcn

The `OutputFcn` property lets you define your own output function and pass the name of this function to the ODE solvers. If no output arguments are specified, the solvers call this function after each successful time step. You can use this feature, for example, to plot results as they are computed.

You must code your output function in a specific way for it to interact properly with the ODE solvers. When the name of an executable M-file function, e.g., `myfun`, is passed to an ODE solver as the `OutputFcn` property,

```matlab
options = odeset('OutputFcn', 'myfun')
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputFcn</td>
<td>String</td>
<td>The name of an output function.</td>
</tr>
<tr>
<td>OutputSel</td>
<td>Vector of indices</td>
<td>Indices of solver output components to pass to an output function.</td>
</tr>
<tr>
<td>Refine</td>
<td>Positive integer</td>
<td>Produces smoother output, increasing the number of output points by a factor of <code>Refine</code>. If <code>Refine</code> is 1, the solver returns solutions only at the end of each time step. If <code>Refine</code> is <code>n &gt; 1</code>, the solver uses continuous extension to subdivide each time step into <code>n</code> smaller intervals, and returns solutions at each time point. <code>Refine</code> is 1 by default in all solvers except <code>ode45</code> where it is 4 because of the solver's large step sizes. <code>Refine</code> does not apply when <code>length(tspan)</code> &gt; 2.</td>
</tr>
<tr>
<td>Stats</td>
<td>on</td>
<td>Specifies whether statistics about the solver's computations should be displayed.</td>
</tr>
</tbody>
</table>

**Stats**

Specifies whether statistics about the solver’s computations should be displayed.
the solver calls it with `myfun(tspan, y0,'init')` before beginning the integration so that the output function can initialize. Subsequently, the solver calls `statu = myfun(t,y)` after each step. In addition to your intended use of `(t,y)`, code `myfun` so that it returns a `statu` output value of 0 or 1. If `statu = 1`, integration halts. This might be used, for instance, to implement a **STOP** button. When integration is complete, the solver calls the output function with `myfun([ ], [ ], 'done')`.

Some example output functions are included with the ODE solvers:
- `odeplot` – time series plotting
- `odephas2` – two-dimensional phase plane plotting
- `odephas3` – three-dimensional phase plane plotting
- `odeprint` – print solution as it is computed

Use these as models for your own output functions. `odeplot` is the default output function for all the solvers. It is automatically invoked when the solvers are called with no output arguments.

**OutputSel**

The **OutputSel** property is a vector of indices specifying which components of the solution vector are to be passed to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using

```
options = odeset('OutputFcn','odeplot','OutputSel',[1 3]);
```

**Refine**

The **Refine** property, an integer `n`, produces smoother output by increasing the number of output points by a factor of `n`. This feature is especially useful when using a medium or high order solver, such as `ode45`, for which solution components can change substantially in the course of a single step. To obtain smoother plots, increase the **Refine** property.

**NOTE** In all the solvers, the default value of **Refine** is 1. Within `ode45`, however, **Refine** is 4 to compensate for the solver’s large step sizes. To override this and see only the time steps chosen by `ode45`, set **Refine** to 1.
The extra values produced for *Refine* are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

**Stats**

The `Stats` property specifies whether statistics about the computational cost of the integration should be displayed. By default, `Stats` is off. If it is on, after solving the problem the integrator displays:

- The number of successful steps.
- The number of failed attempts.
- The number of times the ODE file was called to evaluate \( F(t,y) \).
- The number of times that the partial derivatives matrix \( \frac{\partial F}{\partial y} \) was formed.
- The number of LU decompositions.
- The number of solutions of linear systems.

You can obtain the same values by including a third output argument in the call to the ODE solver:

```matlab
[ T, Y, S ] = ode45('myfun', ...);
```

This statement produces a vector `S` that contains these statistics.

**Jacobian Matrix Properties**

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix \( \frac{\partial F}{\partial y} \), a matrix of partial derivatives of the function defining the differential equation:
There are two aspects to providing information about the Jacobian:

• You can set up your ODE file to calculate and return the value of the Jacobian matrix for the problem. In this case, you must also use `odeset` to set the `Jacobian` property.

• If you do not calculate the Jacobian in the ODE file, `ode15s` and `ode23s` call the helper function `numjac` to approximate Jacobians numerically by finite differences. In this case, you may be able to use the `JConstant`, `Vectorized`, or `JPattern` properties.

The Jacobian matrix properties pertain only to the stiff solvers `ode15s` and `ode23s` for which the Jacobian matrix $\frac{\partial F}{\partial y}$ is critical to reliability and efficiency.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>J Constant</td>
<td>on / off</td>
<td>Set on if the Jacobian matrix $\frac{\partial F}{\partial y}$ is constant (does not depend on $t$ or $y$).</td>
</tr>
<tr>
<td>Jacobian</td>
<td>on / off</td>
<td>Set on to inform the solver that the ODE file is coded such that $F(t,y,'\text{Jacobian}')$ returns $\frac{\partial F}{\partial y}$.</td>
</tr>
<tr>
<td>JPattern</td>
<td>on / off</td>
<td>Set on if $\frac{\partial F}{\partial y}$ is a sparse matrix and the ODE file is coded so that $F([],[],'\text{Pattern}')$ returns a sparsity pattern matrix.</td>
</tr>
<tr>
<td>Vectorized</td>
<td>on / off</td>
<td>Set on to inform the stiff solver that the ODE file is coded so that $F(t,[y_1 y_2 ...])$ returns $[F(t,y_1) F(t,y_2) ...]$.</td>
</tr>
</tbody>
</table>

**J Constant**

Set `J Constant` on if the Jacobian matrix $\frac{\partial F}{\partial y}$ is constant (does not depend on $t$ or $y$). Whether computing the Jacobians numerically or evaluating them
analytically, the solver takes advantage of this information to reduce solution time. For the stiff van der Pol example, the Jacobian matrix is

\[
J = \begin{bmatrix}
0 & 1 \\
-2000*y(1)*y(2) - 1 & 1000*(1-y(1)^2)
\end{bmatrix}
\]

(not constant) so the J Constant property does not apply.

**Jacobian**

Set Jacobian on to inform the solver that the ODE file is coded such that \(F(t, y, 'Jacobian')\) returns \(\partial F / \partial y\). By default, Jacobian is off, and Jacobians are generated numerically.

Coding the ODE file to evaluate the Jacobian analytically often increases the speed and reliability of the solution for the stiff problem. The Jacobian shown above for the stiff van der Pol problem can be coded into the ODE file as

```matlab
function out1 = vdp1000(t,y,flag)
if nargin < 3 | isempty(flag)    % return dy
    out1 = [y(2); 1000*(1–y(1)^2)∗y(2)–y(1)];
elseif strcmp(flag,'jacobian')   % return J
    out1 = [ 0                      1
              (–2000*y(1)*y(2) – 1)  (1000*(1–y(1)^2)) ];
end
```

**JPattern**

Set JPattern on if \(\partial F / \partial y\) is a sparse matrix and the ODE file is coded so that \(F([], [], 'JPattern')\) returns a sparsity pattern matrix. This is a sparse matrix with 1s where there are nonzero entries in the Jacobian. numjac uses the sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large (size greater than approximately 100-by-100) and sparse, this can accelerate execution greatly. For an example using the JPattern property, see the brussode example on page 8-36.

**Vectorized**

Set Vectorized on to inform the stiff solver that the ODE file is coded so that \(F(t, [y_1, y_2, ...])\) returns \([F(t, y_1), F(t, y_2), ...]\). When computing Jacobians numerically, the solver passes this information to the numjac routine. This allows numjac to reduce the number of function evaluations.
required to compute all the columns of the Jacobian matrix, and may reduce solution time significantly.

With MATLAB’s array notation, it is typically an easy matter to vectorize an ODE file. For example, the stiff van der Pol example shown previously can be vectorized by introducing colon notation into the subscripts and by using the array power (.^) and array multiplication (.*) operators:

```matlab
function dy = vdp1000(t, y)
    dy = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

### Step Size Properties

The step size properties let you specify the first step size tried by the solver, potentially helping it to recognize better the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxStep</td>
<td>Positive scalar</td>
<td>Upper bound on solver step size.</td>
</tr>
<tr>
<td>InitialStep</td>
<td>Positive scalar</td>
<td>Suggested initial step size.</td>
</tr>
</tbody>
</table>

Generally it is not necessary for you to adjust MaxStep and InitialStep because the ODE solvers implement state-of-the-art variable time step control algorithms. Adjusting these properties without good reason may result in degraded solver performance.

### MaxStep

MaxStep has a positive scalar value. This property sets an upper bound on the magnitude of the step size the solver uses. If the differential equation has periodic coefficients or solution, it may be a good idea to set MaxStep to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest.

- Do not reduce MaxStep to produce more output points. This can slow down solution time significantly. Instead, use Refine (page 8-24) to compute additional outputs by continuous extension at very low cost.
- Do not reduce MaxStep when the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance RelTol, and use the...
solution you just computed to determine appropriate values for the absolute error tolerance vector \textit{AbsTol}. (See “Error Tolerance Properties” on page 8-21 for a description of the error tolerance properties.)

- Generally you should not reduce \textit{MaxStep} to make sure that the solver doesn’t step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solvers twice. If you do not know the time at which the change occurs, try reducing the error tolerances \textit{RelTol} and \textit{AbsTol}. Use \textit{MaxStep} as a last resort.

\textbf{InitialStep}

\textit{InitialStep} has a positive scalar value. This property sets an upper bound on the magnitude of the first step size the solver tries. Generally the automatic procedure works very well. However, the initial step size is based on the slope of the solution at the initial time \textit{tspan}(1), and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable \textit{InitialStep}.

\textbf{Mass Matrix Properties}

In addition to solving problems of the form \( y' = F(t, y) \), the stiff solvers can solve problems with mass matrices. In particular, \textit{ode15s} can solve problems of the form \( M(t)y' = F(t, y) \) with a mass matrix \( M(t) \) that is nonsingular and (usually) sparse. \textit{ode23s} is limited to problems with a constant matrix \( M \). The mass matrix properties let you specify information about the mass matrix \( M \) for these types of problems.
The nonstiff solvers can’t take special advantage of the form of the mass matrix problem. For these solvers, absorb matrix \( M \) into the definition of \( F \).

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mass</td>
<td>on</td>
<td>Set on to inform the solver that the ODE file is coded such that ( F(t, y, \text{'mass'}) ) returns mass matrix ( M(t) ).</td>
</tr>
<tr>
<td>MassConstant</td>
<td>on</td>
<td>Set on if the mass matrix ( M ) is constant (does not depend on ( t )).</td>
</tr>
</tbody>
</table>

**Mass**

Set \texttt{Mass} on to inform the solver that the ODE file is coded so that \( F(t, y, \text{'mass'}) \) returns mass matrix \( M(t) \). By default, \texttt{Mass} is off.

**MassConstant**

Set \texttt{MassConstant} on to inform the solver that the mass matrix \( M \) is constant (that is, it does not depend on \( t \)). This is relevant only to the \texttt{ode15s} solver, which uses the matrix to reduce solution time. The default value of \texttt{MassConstant} is \texttt{off} in \texttt{ode15s} and is (necessarily) \texttt{on} in \texttt{ode23s}.

For an example of an ODE file with a mass matrix, see “Example 4: Finite Element Discretization” on page 8-40.

**Event Location Property**

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth, or the times at which the ODE solution reaches certain values. While solving a problem, the MATLAB ODE solvers can locate transitions to, from, or through zeros of a vector of user-defined functions.

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>on</td>
<td>Set this on if the ODE file evaluates and returns the event functions, and returns information about the events.</td>
</tr>
</tbody>
</table>
Events

Set this parameter on to inform the solver that the ODE file is coded so that \( F(t, y, 'events') \) returns appropriate event function information. By default, 'events' is off.

For example, the statement

\[
[T, Y, TE, YE, IE] = solver('F', tspan, y0, options)
\]

with the Events property in options set on solves an ODE problem while also locating zero crossings of an events function defined in the ODE file. In this case, the solver returns three additional outputs:

- \( TE \) is a column vector of times at which events occur.
- Rows of \( YE \) are solutions corresponding to times in \( TE \).
- Indices in vector \( IE \) specify which event occurred at the time in \( TE \).

The ODE file must be coded to return three values in response to the 'events' flag,

\[
[value, isterminal, direction] = F(t, y, 'events');
\]

The first output argument \( value \) is the vector of event functions evaluated at \((t, y)\). The \( value \) vector may be any length. It is evaluated at the beginning and end of each integration step, and if any elements make transitions to, from, or through zero (with the directionality specified in constant vector \( direction \)), the solver uses the continuous extension formulas to determine the time when the transition occurred.

Terminal events halt the integration. The argument \( isterminal \) is a logical vector of 1s and 0s that specifies whether a zero-crossing of the corresponding \( value \) element is terminal. 1 corresponds to a terminal event, halting the integration; 0 corresponds to a nonterminal event.

The \( direction \) vector specifies a desired directionality: positive (1), negative (-1), or don't care (0), for each \( value \) element.

The time an event occurs is located to machine precision within an interval of \([t - t+]\). Nonterminal events are reported at \( t + \). For terminal events, both \( t - \) and \( t + \) are reported.

For an example of an ODE file with event location, see “Example 5: Simple Event Location” on page 8-42.
ode15s Properties

The ode15s solver is a variable-order stiff solver based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The ode15s properties let you choose between these formulas, as well as specifying the maximum order for the solver.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxOrder</td>
<td>1</td>
<td>The maximum order formula used.</td>
</tr>
<tr>
<td>BDF</td>
<td>on</td>
<td>Specifies whether the backward differentiation formulas are to be used instead of the default numerical differentiation formulas.</td>
</tr>
</tbody>
</table>

**MaxOrder**

MaxOrder is an integer 1 through 5 used to set an upper bound on the order of the formula that computes the solution. By default, the maximum order is 5.

**BDF**

Set BDF on to have ode15s use the BDFs. By default, BDF is off, and the solver uses the NDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if MaxOrder is reduced (for example to 2) so that only the most stable formulas are used.
Examples: Applying the ODE Solvers

This section contains several examples of ODE files. These examples illustrate the kinds of problems you can solve in MATLAB. For more examples, see MATLAB’s demos directory.

Example 1: Simple Nonstiff Problem

`rigode` is a nonstiff example that can be solved with all five solvers of the ODE suite. It is a standard test problem, proposed by Krogh, for nonstiff solvers. The analytical solutions are Jacobian elliptic functions accessible in MATLAB. The interval here is about 1.5 periods.

The `rigode` system consists of the Euler equations of a rigid body without external forces as proposed by Krogh. `rigode` is a system of three equations:

\[
\begin{align*}
    y_1 &= y_2 y_3 \\
    y_2 &= -y_1 y_3 \\
    y_3 &= -0.51 y_1 y_2
\end{align*}
\]

`rigode([],[],'init')` returns the default `tspan`, `y0`, and `options` values for this problem. These values are retrieved by an ODE solver if the solver is invoked with empty `tspan` or `y0` arguments. This example uses the default solver options, so the third output argument is set to empty, `[]`, instead of an options structure created with `odeset`. By means of the `'init'` flag, the entire initial value problem is defined in one file.
function [out1, out2, out3] = rigidode(t, y, flag)
% RIGIDOIDE Euler equations of a rigid body without external forces.

if nargin < 3 | isempty(flag)
    % Return dy/dt = f(t,y).
    out1 = [y(2) * y(3); -y(1) * y(3); -0.51 * y(1) * y(2)];
else
    switch(flag)
        case 'init'
            % Used only if TSPAN or Y0 is empty.
            % Return default TSPAN, Y0, and OPTIONS.
            out1 = [0; 12];
            out2 = [0; 1; 1];
            out3 = [];
        otherwise
            error(['Unknown flag ''' flag '''']);
    end
end

Example 2: van der Pol Equation

vdpode is a more general version of the van der Pol example that has been used in various forms throughout this chapter. For illustrative purposes, it is coded for both fast numerical Jacobian computation (Vectorized property) and for analytical Jacobian evaluation (Jacobian property). In practice you would supply only one or the other of these options. It is not necessary to supply either.

The van der Pol equation is written as a system of two equations:

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= \mu(1-y_1^2)y_2 - y_1
\end{align*}
\]

vdpode(t, y) or vdpode(t, y, [], mu) returns the derivatives vector for the van der Pol equation. By default, \(\mu\) is 1 and the problem is not stiff. Optionally, pass in the \(\mu\) parameter as an additional input argument to an ODE solver.
The problem becomes more stiff as \( \mu \) is increased and the period of oscillation becomes larger.

When \( \mu \) is 1000 the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff (quasi-discontinuities).

This example sets Vectorized on with odeset because vdpode is coded so that vdpode(\( t, [y_1 y_2 \ldots] \)) returns [vdpode(\( t, y_1 \)) vdpode(\( t, y_2 \)) \ldots] for scalar time \( t \) and vectors \( y_1, y_2, \ldots \). The stiff ODE solvers take advantage of this feature only when approximating the columns of the Jacobian numerically.

vdpode([],[],'init') returns the default tspan, y0, and options values for this problem. The entire initial value problem is defined in this one file.

vdpode(\( t, y, 'j acobi an' \)) or vdpode(\( t, y, 'j acobi an', \mu \)) returns the Jacobian matrix \( \frac{\partial F}{\partial y} \) evaluated analytically at \( (t, y) \). By default, the stiff solvers of the ODE suite approximate Jacobian matrices numerically. However, if Jacobian is set on with odeset, a solver calls the ODE file with the flag 'jacobian' to obtain \( \frac{\partial F}{\partial y} \). Providing the solvers with an analytic Jacobian is not necessary, but it can improve the reliability and efficiency of integration.

```plaintext
function [out1, out2, out3] = vdpode(t, y, flag, mu)
end
```

Examples: Applying the ODE Solvers
else
    switch(flag)
    case 'init'
        % Used only if TSPAN or Y0 is empty.
        % Return default TSPAN, Y0, and OPTIONS.
        out1 = [0; max(20, 3*mu)]; % several periods
        out2 = [2; 0];
        out3 = odeset('Vectorized',1);
    case 'jacobian'
        % Used only if odeset('Jacobian',1).
        % Return Jacobian J(t,y) = df/dy, evaluating it analytically.
        out1 = [1
                (-2*mu*y(1)+y(2) - 1) (mu*(1-y(1)^2)) ];
    otherwise
        error(['Unknown flag '' flag ''']);
    end
end

**Example 3: Large, StiffSparse Problem**

This is an example of a (potentially) large stiff sparse problem. Like vdpode, the file is coded to use both the Vectorized and Jacobian properties, but only one is used during the course of a simulation. Like both previous examples, brussode responds to the 'init' flag.

The brussode example is the classic “Brusselator” system (Hairer and Wanner) modeling diffusion in a chemical reaction,

\[
\begin{align*}
    u_i' &= 1 + u_i^2 v_i - 4 u_i + \alpha(N + 1)^2 (u_{i-1} - 2u_i + u_{i+1}) \\
    v_i' &= 3u_i - u_i^2 v_i + \alpha(N + 1)^2 (v_{i-1} - 2v_i + v_{i+1})
\end{align*}
\]

and is solved on the time interval \([0, 10]\) with \(\alpha = 1/50\) and

\[
\begin{align*}
    u_i(0) &= 1 + \sin(2\pi x_i) \\
    v_i(0) &= 3
\end{align*}
\]

with \(x_i = i/(N + 1)\) for \(i = 1, \ldots, N\).

There are \(2N\) equations in the system, but the Jacobian is banded with a constant width 5 if the equations are ordered as \(u_1, v_1, u_2, v_2, \ldots\)
brussode(t, y) or brussode(t, y, [], n) returns the derivatives vector for the Brusselator problem. The parameter \( n \geq 2 \) is used to specify the number of grid points; the resulting system consists of \( 2n \) equations. By default, \( n \) is 2. The problem becomes increasingly stiff and the Jacobian increasingly sparse as \( n \) is increased.

brussode([], [], 'jpattern') or brussode([], [], 'jpattern', n) returns a sparse matrix of 1s and 0s showing the locations of nonzeros in the Jacobian \( \partial F/\partial y \). By default, the stiff ODE solvers generate Jacobians numerically as full matrices. However, if JPattern is set on with odeset, a solver calls the ODE file with the flag 'jpattern'. This provides the solver with a sparsity pattern that it uses to generate the Jacobian numerically as a sparse matrix. Providing a sparsity pattern can significantly reduce the number of function evaluations required to generate the Jacobian and can accelerate integration. For the Brusselator problem, if the sparsity pattern is not supplied, \( 2n \) evaluations of the function are needed to compute the \( 2n \)-by-\( 2n \) Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of \( n \).

function [out1, out2, out3] = brussode(t, y, flag, N)
% BRUSSODE Stiff problem modeling a chemical reaction.

if nargin < 4 | isempty(N)
    N = 2;
end

if nargin < 3 | isempty(flag)
    % Return dy/dt = f(t, y).
    c = 0.02 * (N+1)^2;
    out1 = zeros(2*N, size(y, 2));    % preallocate dy/dt, 'vectorized'
    % Evaluate the 2 components of the function at one edge of the grid
    i = 1;
    out1(i,:) = 1 + y(i+1,:) .* y(i,:) .^ 2 - 4 .* y(i,:) + ... 
                c .* (1-2.*y(i,:) + y(i+2,:));
    out1(i+1,:) = 3 .* y(i,:) - y(i+1,:) .* y(i,:) .^ 2 + ... 
                c .* (3-2.*y(i+1,:) + y(i+3,:));

    % Evaluate both components of the function at interior grid pts.
    i = 3:2:2*N-3;
    out1(i,:) = 1 + y(i+1,:) .* y(i,:) .^ 2 - 4 .* y(i,:) + ... 
                c .* (y(i-2,:) - 2.*y(i,:) + y(i+2,:));
    out1(i+1,:) = 3 .* y(i,:) - y(i+1,:) .* y(i,:) .^ 2 + ... 
                c .* (y(i-1,:) - 2.*y(i+1,:) + y(i+3,:));

    % Evaluate the 2 components of the function at the other edge of the grid (with edge conditions).
    i = 2*N-1;
    out1(i,:) = 1 + y(i+1,:) .* y(i,:) .^ 2 - 4 .* y(i,:) + ... 
                c .* (y(i-2,:) - 2.*y(i,:) + 1);
    out1(i+1,:) = 3 .* y(i,:) - y(i+1,:) .* y(i,:) .^ 2 + ... 
                c .* (y(i-1,:) - 2.*y(i+1,:) + 3);
Examples: Applying the ODE Solvers

else
  switch(flag)
    case 'init' % Used only if TSPAN or Y0 is empty.
      % Return default TSPAN, Y0, and OPTIONS.
      out1 = [0; 10];
      out2 = [1+sin((2*pi/(N+1))*(1:N)); 3+zeros(1,N)];
      out2 = out2(:);
      out3 = odeset('Vectorized','on');
    case 'jacobian' % Used only if odeset('Jacobian','on')
      % Return Jacobian matrix J(t,y) = df/dy, evaluating it
      % analytically.
      c = 0.02 * (N+1)^2;
      B = zeros(2*N, 5);
      B(1:2*(N-1), 1) = B(1:2*(N-1), 1) + c;
      i = 1:2:2*N-1;
      B(i, 2) = 3 - 2*y(i) .* y(i+1);
      B(i, 3) = 2*y(i) .* y(i+1) - 4 - 2*c;
      B(i+1, 3) = -y(i) .* 2 - 2*c;
      B(i+1, 4) = y(i) .* 2;
      B(3:2*N, 5) = B(3:2*N, 5) + c;
      out1 = spdiags(B, -2:2, 2*N, 2*N); % This is a SPARSE Jacobian.
    case 'jpattern' % Used only if odeset('JPattern','on')
      % and generating Jacobian numerically.
      % Return sparsity pattern S.
      B = ones(2*N, 5);
      B(2:2:2*N, 2) = zeros(N, 1);
      B(1:2:2*N-4, 4) = zeros(N, 1);
      out1 = spdiags(B, -2:2, 2*N, 2*N);
    otherwise
      error(['Unknown flag ''flag''']);
  end
end
Example 4: Finite Element Discretization

fem1ode(t, y) or fem1ode(t, y, [], n) returns the derivatives vector for a finite element discretization of a partial differential equation. The parameter n controls the discretization, and the resulting system consists of n equations. By default, n is 9.

This example involves a mass matrix. The system of ODE’s comes from a method of lines solution of the partial differential equation

\[ e^{-t} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \]

with initial condition \( u(0, x) = \sin(x) \) and boundary conditions \( u(t, 0) = u(t, \pi) = 0 \). An integer N is chosen, \( h \) is defined as \( 1/(N+1) \), and the solution of the partial differential equation is approximated at \( x_k = k\pi h \) for \( k = 0, 1, ..., N+1 \) by

\[
 u(t,x_k) = \sum_{k=1}^{N} c_k(t)\phi_k(x)
\]

Here \( \phi_k(x) \) is a piecewise linear function that is 1 at \( x_k \) and 0 at all the other \( x_j \). A Galerkin discretization leads to the system of ODE’s

\[
 A(t)c' = Rc \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ \vdots \\ c_N(t) \end{bmatrix}
\]

and the tridiagonal matrices \( A(t) \) and \( R \) are given by

\[
 A_{ij} = \begin{cases} 
 \exp(-t)2h/3 & \text{if } i = j \\
 \exp(-t)h/6 & \text{if } i = j \pm 1 \\
 0 & \text{otherwise}
\end{cases}
\]

and

\[
 R_{ij} = \begin{cases} 
 -2/h & \text{if } i = j \\
 1/h & \text{if } i = j \pm 1 \\
 0 & \text{otherwise}
\end{cases}
\]

The initial values \( c(0) \) are taken from the initial condition for the partial differential equation. The problem is solved on the time interval \([0, \pi] \).

fem1ode(t, [], 'mass') or fem1ode(t, [], 'mass', n) returns the time-dependent mass matrix \( M \) evaluated at time \( t \). By default, \texttt{ode15s} solves
systems of the form \( y' = F(t, y) \). However, if the Mass property is set on with odeset, the solver calls the ODE file with the flag 'mass'. This provides the solver with a mass matrix that it uses to solve \( M(t)y' = F(t, y) \). If the mass matrix is a constant \( M \), the problem can be also be solved with ode23s.

For example, to solve a system of 20 equations, use

\[
[T, Y] = \text{ode15s}('fem1ode',[],[],\text{odeset('Mass','on')},20);
\]

fem1ode also responds to the flag 'init' (see the rigidode example for details).

\[
\text{function } [\text{out1}, \text{out2}, \text{out3}] = \text{fem1ode}(t,y,\text{flag},N)
\]

\[
\text{%FEM1ODE Stiff problem with a time-dependent mass matrix,}
\]
\[
\text{%M(t)y' = f(t,y).}
\]

\[
\text{if } \text{numargin} < 4 \text{ } \text{isempty}(N)
\]
\[
N = 9;
\]

\[
\text{end}
\]

\[
\text{if } \text{numargin} < 3 \text{ } \text{isempty}(\text{flag})
\]

\[
\text{\% Return dy/dt = f(t,y).}
\]
\[
e = ((N+1)/\pi) + \text{zeros}(N,1); \quad \text{%pi=pi/(N+1); e=(1/h)+zeros(N,1);}
\]
\[
R = \text{spdiags}([e -2*e e], -1:1, N, N);
\]
\[
\text{out1} = R*y;
\]

\[
\text{else}
\]

\[
\text{switch(\text{flag})}
\]
\[
\text{case 'init' } \text{% Used only if TSPAN or Y0 is empty.}
\]
\[
\text{% Return default TSPAN, Y0, and OPTIONS.}
\]
\[
\text{out1} = [0; pi];
\]
\[
\text{out2} = \text{sin}((\pi/(N+1))*(1:N))';
\]
\[
\text{out3} = \text{odeset('Mass','on','Vectorized','on')};
\]

\[
\text{case 'mass' } \text{% Used only if odeset('Mass','on').}
\]
\[
\text{% Return mass matrix } M(t).
\]
\[
e = (\exp(-t)*\pi/(6*(N+1))) + \text{zeros}(N,1); \quad \text{%pi=pi/(N+1);}
\]
\[
e=\exp(-t)+h/6+\text{zeros}
\]
\[
\text{out1} = \text{spdiags}([e 4*e e], -1:1, N, N);
\]

8-41
Example 5: Simple Event Location

`ballode(t,y)` returns the derivatives vector for the equations of motion of a bouncing ball. This ODE file illustrates the event location capabilities of the ODE solvers.

The equations for the bouncing ball are

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= -9.8
\end{align*}
\]

`ballode(t,y,'events')` returns a zero-crossing vector `value` evaluated at `(t,y)`, as well as two constant vectors `isterminal` and `direction`. By default, the ODE solvers do not locate zero-crossings. However, if the `Events` property is set on with `odeset`, a solver calls the ODE file with the flag `'events'`. This provides the solver with information that it uses to locate zero-crossings of the elements in the `value` vector. The `value` vector may be any length. It is evaluated at the beginning and end of a step, and if any elements change sign (with the directionality specified in `direction`), the zero-crossing point is located. The `isterminal` vector consists of logical 1s and 0s, enabling you to specify whether or not a zero-crossing of the corresponding `value` element halts the integration. The `direction` vector enables you to specify a desired directionality, positive (1), negative (-1), or don't care (0) for each `value` element.

`ballode` also responds to the flag `'init'` (see the `rigidode` example for details).
function [out1, out2, out3] = ballode(t, y, flag)
% BALLODE Equations of motion for a bouncing ball.

if nargin < 3 | isempty(flag)
    % Return dy/dt = f(t, y).
    out1 = [y(2); -9.8];
else
    switch(flag)
    case 'init' % Used only if TSPAN or Y0 is empty.
        % Return default TSPAN, Y0, and OPTIONS.
        out1 = [0; 10];
        out2 = [0; 20];
        out3 = odeset('Events','on');
    case 'events' % Used only if odeset('Events','on').
        % Return event vectors VALUE, ISTERMINAL, and DIRECTION.
        % Locate zero-crossings of both height and velocity.
        out1 = y;    % [height; velocity]
        out2 = [1; 0];   % stop at zeros of height
        out3 = ones(2); % event only when value(1) is decreasing, don't care for
                        % value(2)
    otherwise
        error(['Unknown flag ''flag''']);
    end
end

Example 6: Advanced Event Location
orbitode is a standard test problem for nonstiff solvers presented in Shampine and Gordon, (see reference that follows).
The orbitode problem is a system of four equations:

\[ y_1' = y_3 \]
\[ y_2' = y_4 \]
\[ y_3' = 2y_4 + y_1 - \frac{\mu^*(y_1 + \mu)}{r_1^3} - \frac{\mu(y_1 - \mu^*)}{r_2^3} \]
\[ y_4' = -2y_3 + y_2 - \frac{\mu^*y_2}{r_1^3} - \frac{\mu y_2}{r_2^3} \]

where

\[ \mu = 1/82.45 \]
\[ \mu^* = 1 - \mu \]
\[ r_1 = \sqrt{(y_1 + \mu)^2 + y_2^2} \]
\[ r_2 = \sqrt{(y_1 - \mu^*)^2 + y_2^2} \]

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body around the other two bodies. The initial conditions have been chosen so as to make the orbit periodic. This corresponds to a spaceship traveling around the moon and returning to the earth. Moderately stringent tolerances are necessary to reproduce the qualitative behavior of the orbit. Suitable values are \(1e-5\) for \(\text{RelTol}\) and and \(1e-4\) for \(\text{AbsTol}\).

The event functions implemented in this example locate the point of maximum distance from the earth and the time the spaceship returns to earth.
function [out1, out2, out3] = orbitode(t, y, flag)
%ORBITODE Restricted 3 body problem used by ORBITDEMO.

if nargin < 3 | isempty(flag)
    % Return dy/dt = f(t, y).
    mu = 1 / 82.45;
    mustar = 1 – mu;
    r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
    r23 = ((y(1) – mustar)^2 + y(2)^2) ^ 1.5;
    out1 = [ y(3),
             y(4),
             (2 * y(4) + y(1) – mustar * ((y(1) + mu) / r13) – ...
               mu * ((y(1) – mustar) / r23))];
else
    y0 = [1.2; 0; 0; –1.04935750983031990726];

    switch(flag)
    case 'init' % Used only if TSPAN or Y0 is empty.
        % Return default TSPAN, Y0, and OPTIONS.
        out1 = [0; 6.19216933131963970674];
        out2 = y0;
        out3 = odeset('RelTol', 1e–5, 'AbsTol', 1e–4);
    case 'events' % Used only if odeset('Events', 'on').
        % Return event vectors VALUE, ISTERMNAL, and DIRCTON.
        % DSQ is the square of the distance from the initial point to the current position of the body:
% DSQ = \((y(1)-y0(1))^2 + (y(2)-y0(2))^2 = \langle y(1:2)-y0, y(1:2)-y0\rangle\%

% Local minimum of DSQ occurs when d/dt DSQ crosses zero heading in
% the positive direction. We can compute d/dt DSQ as
%
% d/dt DSQ = 2*(y(1:2)-y0)'*(dy(1:2)/dt = 2*(y(1:2)-y0)'*y(3:4)
%
% dDSQdt = 2 * ( (y(1:2)-y0(1:2))' * y(3:4) );

out1 = [dDSQdt; dDSQdt]; % Report local minima and local maxima.
out2 = [1; 0]; % Stop at minima, not at maxima.
% dDSQdt is increasing at minima, and decreasing at maxima.
out3 = [1; -1];

otherwise
error(['Unknown flag ''flag'''']);
end
end
Questions and Answers

This section contains a number of tables that answer questions about the use and operation of the MATLAB ODE solvers. The question and answer tables cover six categories of information:

- General ODE Solver Questions
- Problem Size, Memory Use, and Computation Speed
- Time Steps for Integration
- Error Tolerance and Other Options
- Troubleshooting
- Solving Different Kinds of Systems

### General ODE Solver Questions

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do the ODE solvers differ from quad or quad8?</td>
<td>quad and quad8 solve problems of the form ( y' = F(t) ). The ODE suite solves more general problems of the form ( y' = F(t, y) ).</td>
</tr>
<tr>
<td>Can I solve ODE systems in which there are more equations than unknowns, or vice-versa?</td>
<td>No.</td>
</tr>
</tbody>
</table>

### Time Steps for Integration

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first step size that the integrator takes is too large, and it misses important behavior.</td>
<td>You can specify the first step size with the InitialStep property. The integrator tries this value, then reduces it if necessary.</td>
</tr>
<tr>
<td>Can I integrate with fixed step sizes?</td>
<td>No.</td>
</tr>
</tbody>
</table>
Error Tolerance and Other Options

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do I choose RelTol and AbsTol?</td>
<td>RelTol, the relative accuracy tolerance, controls the number of correct digits in the answer. AbsTol, the absolute error tolerance, controls the difference between the answer and the solution. A relative error tolerance gets into trouble when a solution component vanishes. An absolute error tolerance gets into trouble when a solution component is unexpectedly large. The solvers require nonzero tolerances and use a mixed test to avoid these problems. At each step the error $e$ in the $i$th component of the solution is required to satisfy this condition</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>The use of RelTol is clear – to obtain $p$ correct digits let RelTol $= 10^{-p}$, or slightly smaller. The use of AbsTol depends on the problem scale. AbsTol is a threshold – the solver does not guarantee correct digits for solution components smaller than AbsTol($i$). If the problem has a natural threshold, use it as AbsTol. A small value of AbsTol does not adversely affect the computation, but be aware that the problem's scaling might mean that an important component is smaller than the specified AbsTol. You might think that you computed the component with the relative accuracy of RelTol, when in fact it is below the AbsTol threshold, and you have few if any correct digits. Even if you are not interested in correct digits in this component, failing to compute it accurately may harm the accuracy of components you do care about. Generally the solvers handle this situation automatically, but not always.</td>
</tr>
</tbody>
</table>

I want answers that are correct to the precision of the computer. Why can't I simply set RelTol to eps? | You can get close to machine precision, but not that close. The solvers do not allow RelTol near eps because they try to approximate a continuous function. At tolerances comparable to eps, the machine arithmetic causes all functions to look discontinuous. |
### Error Tolerance and Other Options

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do I tell the solver that I don’t care about getting an accurate answer for one of the solution components?</td>
<td>You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.</td>
</tr>
</tbody>
</table>

### Troubleshooting

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>The solution doesn’t look like what I expected.</td>
<td>If you’re right about its appearance, you need to reduce the error tolerances from their default values. A smaller relative error tolerance is needed to compute accurately the solution of problems integrated over “long” intervals, as well as solutions of problems that are moderately unstable. You should check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not asking for any correct digits in these components. This may be acceptable for these components, but failing to compute them accurately may degrade the accuracy of other components that depend on them.</td>
</tr>
<tr>
<td>My plots aren’t smooth enough.</td>
<td>Increase the value of <code>Refine</code> from its default of 4 in <code>ode45</code> and 1 in the other solvers. The bigger the value of <code>Refine</code>, the more output points. Execution speed is not affected much by the value of <code>Refine</code>.</td>
</tr>
<tr>
<td>I’m plotting the solution as it is computed and it looks fine, but the code gets stuck at some point.</td>
<td>First verify that the ODE function is smooth near the point where the code gets stuck. If it isn’t, the solver must take small steps to deal with this. It may help to break <code>tspan</code> into pieces on which the ODE function is smooth. If the function is smooth and the code is taking extremely small steps, you are probably trying to solve a stiff problem with a solver not intended for this purpose. Switch to <code>ode15s</code> or <code>ode23s</code>.</td>
</tr>
</tbody>
</table>
### Troubleshooting

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>My integration proceeds very slowly, using too many time steps.</td>
<td>First, check that your <code>tspan</code> is not too long. Remember that the solver will use as many time points as necessary to produce a smooth solution. If the ODE function changes on a time scale that is very short compared to the <code>tspan</code>, then the solver will use a lot of time steps. Long-time integration is a hard problem. Break <code>tspan</code> into smaller pieces. If the ODE function does not change noticeably on the <code>tspan</code> interval, it could be that your problem is stiff. Try using <code>ode15s</code> or <code>ode23s</code>. Finally, make sure that the ODE function is written in an efficient way. The solvers evaluate the derivatives in the ODE function many times. The cost of numerical integration depends critically on the expense of evaluating the ODE function. Rather than recompute complicated constant parameters every evaluation, store them in globals or calculate them once outside the function and pass them in as additional parameters.</td>
</tr>
<tr>
<td>I know that the solution undergoes a radical change at time <code>t</code> where <code>t0 ≤ t ≤ tfinal</code> but the integrator steps past without “seeing” it.</td>
<td>If you know there is a sharp change at time <code>t</code>, it might help to break the <code>tspan</code> interval into two pieces, <code>[t0, t]</code> and <code>[t, tfinal]</code>, and call the integrator twice. If the differential equation has periodic coefficients or solution, you might restrict the maximum step size to the length of the period so the integrator won’t step over periods.</td>
</tr>
</tbody>
</table>
Solving Different Kinds of Systems

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can the solvers handle PDEs that have been discretized by the Method of Lines?</td>
<td>Yes. What you obtain is a system of ODEs. Depending on the discretization, you might have a form involving mass matrices – ode15s and ode23s provide for this. Often the system is stiff. This is to be expected when the PDE is parabolic and when there are phenomena that happen on very different time scales such as a chemical reaction in a fluid flow. In such cases, use ode15s or ode23s. If, as usual, there are many equations, set the j Pattern property. This is easy and might make the difference between success and failure due to the computation being too expensive. When the system is not stiff, or not very stiff, ode23 or ode45 will be more efficient than ode15s or ode23s.</td>
</tr>
<tr>
<td>Can I solve differential/algebraic (DAE) systems?</td>
<td>Not currently. The mass matrices accepted by ode15s and ode23s must be nonsingular.</td>
</tr>
<tr>
<td>Can I integrate a set of sampled data?</td>
<td>Not directly. You have to represent the data as a function by interpolation or some other scheme for fitting data. The smoothness of this function is critical. A piecewise polynomial fit like a spline can look smooth to the eye, but rough to a solver; the solver will take small steps where the derivatives of the fit have jumps. Either use a smooth function to represent the data or use one of the lower order solvers (ode23 or ode23s) that is less sensitive to this.</td>
</tr>
<tr>
<td>Can I solve delay-differential equations?</td>
<td>Not directly. In some cases it is possible to use the initial value problem solvers to solve delay-differential equations by breaking the simulation interval into smaller intervals the length of a single delay.</td>
</tr>
<tr>
<td>What do I do when I have the final and not the initial value?</td>
<td>ode45 and the other solvers that are available in this version of the MATLAB ODE suite allow you to solve backwards or forwards in time. The syntax for the solvers is [T, Y] = ode45(’ydot’, [t0 t final], y0); and the syntax accepts t0 &gt; t final.</td>
</tr>
</tbody>
</table>
Sparse Matrices

Introduction .......................................................... 9-5
Sparse Matrix Storage .............................................. 9-5
Creating Sparse Matrices ......................................... 9-6
Importing Sparse Matrices from Outside MATLAB .......... 9-10

Viewing Sparse Matrices ............................................ 9-11
General Storage Information ...................................... 9-11
Information About Nonzero Elements ......................... 9-11
Viewing Sparse Matrices Graphically ......................... 9-13
The find Function and Sparse Matrices ...................... 9-14

Example: Adjacency Matrices and Graphs ..................... 9-15
Graphing Using Adjacency Matrices ............................ 9-15
The Bucky Ball ...................................................... 9-16
An Airflow Model .................................................. 9-21

Sparse Matrix Operations .......................................... 9-23
Standard Mathematical Operations ............................ 9-23
Permutation and Reordering ...................................... 9-24
Factorization ......................................................... 9-28
Simultaneous Linear Equations ................................. 9-33
Eigenvalues and Singular Values ............................... 9-36
MATLAB supports sparse matrices, matrices that contain a small proportion of nonzero elements. This characteristic provides advantages in both matrix storage space and computation time.

This chapter explains how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations.

The sparse matrix functions are located in the `sparfun` directory in the MATLAB toolbox directory.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary sparse matrices.</td>
<td>speye</td>
<td>Sparse identity matrix.</td>
</tr>
<tr>
<td></td>
<td>sprand</td>
<td>Sparse uniformly distributed random matrix.</td>
</tr>
<tr>
<td></td>
<td>sprandn</td>
<td>Sparse normally distributed random matrix.</td>
</tr>
<tr>
<td></td>
<td>sprandsym</td>
<td>Sparse random symmetric matrix.</td>
</tr>
<tr>
<td></td>
<td>spdiags</td>
<td>Sparse matrix formed from diagonals.</td>
</tr>
<tr>
<td>Full to sparse conversion.</td>
<td>sparse</td>
<td>Create sparse matrix.</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>Convert sparse matrix to full matrix.</td>
</tr>
<tr>
<td></td>
<td>find</td>
<td>Find indices of nonzero elements.</td>
</tr>
<tr>
<td></td>
<td>spconvert</td>
<td>Import from sparse matrix external format.</td>
</tr>
<tr>
<td>Working with sparse matrices.</td>
<td>nnz</td>
<td>Number of nonzero matrix elements.</td>
</tr>
<tr>
<td></td>
<td>nonzeros</td>
<td>Nonzero matrix elements.</td>
</tr>
<tr>
<td></td>
<td>nzmax</td>
<td>Amount of storage allocated for nonzero matrix elements.</td>
</tr>
<tr>
<td></td>
<td>spones</td>
<td>Replace nonzero sparse matrix elements with ones.</td>
</tr>
<tr>
<td></td>
<td>spalloc</td>
<td>Allocate space for sparse matrix.</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>issparse</td>
<td>True for sparse matrix.</td>
</tr>
<tr>
<td></td>
<td>spfun</td>
<td>Apply function to nonzero matrix elements.</td>
</tr>
<tr>
<td></td>
<td>spy</td>
<td>Visualize sparsity pattern.</td>
</tr>
<tr>
<td></td>
<td>gplot</td>
<td>Plot graph, as in “graph theory.”</td>
</tr>
<tr>
<td>Reordering algorithms.</td>
<td>colmd</td>
<td>Column minimum degree permutation.</td>
</tr>
<tr>
<td></td>
<td>symmd</td>
<td>Symmetric minimum degree permutation.</td>
</tr>
<tr>
<td></td>
<td>symrcm</td>
<td>Symmetric reverse Cuthill-McKee permutation.</td>
</tr>
<tr>
<td></td>
<td>colperm</td>
<td>Column permutation.</td>
</tr>
<tr>
<td></td>
<td>randperm</td>
<td>Random permutation.</td>
</tr>
<tr>
<td></td>
<td>dmperm</td>
<td>Dulmage-Mendelsohn permutation.</td>
</tr>
<tr>
<td>Linear algebra.</td>
<td>eig</td>
<td>A few eigenvalues.</td>
</tr>
<tr>
<td></td>
<td>svds</td>
<td>A few singular values.</td>
</tr>
<tr>
<td></td>
<td>luinc</td>
<td>Incomplete LU factorization.</td>
</tr>
<tr>
<td></td>
<td>cholinc</td>
<td>Incomplete Cholesky factorization.</td>
</tr>
<tr>
<td></td>
<td>normest</td>
<td>Estimate the matrix 2-norm.</td>
</tr>
<tr>
<td></td>
<td>condest</td>
<td>1-norm condition number estimate.</td>
</tr>
<tr>
<td></td>
<td>sprank</td>
<td>Structural rank.</td>
</tr>
<tr>
<td>Linear equations (iterative</td>
<td>pcg</td>
<td>Preconditioned Conjugate Gradients Method.</td>
</tr>
<tr>
<td>methods).</td>
<td>bi cg</td>
<td>BiConjugate Gradients Method.</td>
</tr>
<tr>
<td></td>
<td>bi cgst ab</td>
<td>BiConjugate Gradients Stabilized Method.</td>
</tr>
</tbody>
</table>
## Sparse Matrices

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cgs</td>
<td>Conjugate Gradients Squared Method.</td>
</tr>
<tr>
<td></td>
<td>gmres</td>
<td>Generalized Minimum Residual Method.</td>
</tr>
<tr>
<td></td>
<td>qmr</td>
<td>Quasi-Minimal Residual Method.</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>symbfact</td>
<td>Symbolic factorization analysis.</td>
</tr>
<tr>
<td></td>
<td>spparms</td>
<td>Set parameters for sparse matrix routines.</td>
</tr>
<tr>
<td></td>
<td>spaugment</td>
<td>Form least squares augmented system.</td>
</tr>
</tbody>
</table>
Introduction

Sparse matrices are a special class of matrices that contain a significant number of zero-valued elements. This property allows MATLAB to

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements.

Sparse Matrix Storage

For full matrices, MATLAB stores internally every matrix element. Zero-valued elements require the same amount of storage space as any other matrix element. For sparse matrices, however, MATLAB stores only the nonzero elements and their indices. For large matrices with a high percentage of zero-valued elements, this scheme significantly reduces the amount of memory required for data storage.

MATLAB uses three arrays internally to store sparse matrices with real elements. Consider an m-by-n sparse matrix with \( nnz \) nonzero entries:

- The first array contains all the nonzero elements of the array in floating-point format. The length of this array is equal to \( nnz \).
- The second array contains the corresponding integer row indices for the nonzero elements. This array also has length equal to \( nnz \).
- The third array contains integer pointers to the start of each column. This array has length equal to \( n \).

This matrix requires storage for \( nnz \) floating-point numbers and \( nnz + n \) integers. At 8 bytes per floating-point number and 4 bytes per integer, the total number of bytes required to store a sparse matrix is

\[
8 \times nnz + 4 \times (nnz + n)
\]

Sparse matrices with complex elements are also possible. In this case, MATLAB uses a fourth array with \( nnz \) elements to store the imaginary parts of the nonzero elements. An element is considered nonzero if either its real or imaginary part is nonzero.
Creating Sparse Matrices
MATLAB never creates sparse matrices automatically. Instead, you must
determine if a matrix contains a large enough percentage of zeros to benefit
from sparse techniques.

The density of a matrix is the number of nonzero elements divided by the total
number of matrix elements. Matrices with very low density are often good
candidates for use of the sparse format.

Converting Full to Sparse
You can convert a full matrix to sparse storage using the `sparse` function with
a single argument.

```matlab
S = sparse(A)
```

For example

```matlab
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0 ];

S = sparse(A)
```

produces

```matlab
S =

(3,1)        1
(2,2)        2
(3,2)        3
(4,3)        4
(1,4)        5
```

The printed output lists the nonzero elements of S, together with their row and
column indices. The elements are sorted by columns, reflecting the internal
data structure.

You can convert a sparse matrix to full storage using the `full` function,
provided the matrix order is not too large. For example

```matlab
A = full(S)
```
Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

**Creating Sparse Matrices Directly**

You can create a sparse matrix from a list of nonzero elements using the `sparse` function with five arguments:

\[
S = \text{sparse}(i, j, s, m, n)
\]

\(i\) and \(j\) are vectors of row and column indices, respectively, for the nonzero elements of the matrix. \(s\) is a vector of nonzero values whose indices are specified by the corresponding \((i, j)\) pairs. \(m\) is the row dimension for the resulting matrix, and \(n\) is the column dimension.

The matrix \(S\) of the previous example can be generated directly with

\[
S = \text{sparse}([3 2 3 4], [1 2 2 3], [1 2 3 4 5], 4, 4)
\]

\[
S =
\begin{pmatrix}
(3, 1) & 1 \\
(2, 2) & 2 \\
(3, 2) & 3 \\
(4, 3) & 4 \\
(1, 4) & 5
\end{pmatrix}
\]

The `sparse` command has a number of alternate forms. The example above uses a form that sets the maximum number of nonzero elements in the matrix to \(\text{length}(s)\). If desired, you can append a sixth argument that specifies a larger maximum, allowing you to add nonzero elements later without changing storage requirements.

**Example: The Second Difference Operator**

The matrix representation of the second difference operator is a good example of a sparse matrix. It is a tridiagonal matrix with –2s on the diagonal and 1s
on the super- and subdiagonal. There are many ways to generate it – here’s one possibility:

\[
D = \text{sparse}(1:n, 1:n, -2*ones(1, n), n, n);
E = \text{sparse}(2:n, 1:n-1, ones(1, n-1), n, n);
S = E + D + E'
\]

For \(n = 5\), MATLAB responds with

\[
S = \\
(1, 1) & -2 \\
(2, 1) & 1 \\
(1, 2) & 1 \\
(2, 2) & -2 \\
(3, 2) & 1 \\
(2, 3) & 1 \\
(3, 3) & -2 \\
(4, 3) & 1 \\
(3, 4) & 1 \\
(4, 4) & -2 \\
(5, 4) & 1 \\
(4, 5) & 1 \\
(5, 5) & -2 \\
\]

Now \(F = \text{full}(S)\) displays the corresponding full matrix.

\[
F = \text{full}(S)
\]

\[
F = \\
-2 & 1 & 0 & 0 & 0 \\
1 & -2 & 1 & 0 & 0 \\
0 & 1 & -2 & 1 & 0 \\
0 & 0 & 1 & -2 & 1 \\
0 & 0 & 0 & 1 & -2 \\
\]

---

**Creating Sparse Matrices from Their Diagonal Elements**

Creating sparse matrices based on their diagonal elements is a common operation, so the function \text{spdiags} handles this task. Its syntax is

\[
S = \text{spdiags}(B, d, m, n)
\]
To create an output matrix $S$ of size $m$-by-$n$ with elements on $p$ diagonals:

- $B$ is a matrix of size $\min(m, n)$-by-$p$. The columns of $B$ are the values to populate the diagonals of $S$.
- $d$ is a vector of length $p$ whose integer elements specify which diagonals of $S$ to populate.

That is, the elements in column $j$ of $B$ fill the diagonal specified by element $j$ of $d$. As an example, consider the matrix $B$ and the vector $d$:

$$B = \begin{bmatrix} 41 & 11 & 0 \\ 52 & 22 & 0 \\ 63 & 33 & 13 \\ 74 & 44 & 24 \end{bmatrix}$$

$$d = \begin{bmatrix} -3 \\ 0 \\ 2 \end{bmatrix}$$

Use these matrices to create a 7-by-4 sparse matrix $A$:

$$A = \text{spdiags}(B, d, 7, 4)$$

$$A = \begin{bmatrix} (1, 1) & 11 \\ (4, 1) & 41 \\ (2, 2) & 22 \\ (5, 2) & 52 \\ (1, 3) & 13 \\ (3, 3) & 33 \\ (6, 3) & 63 \\ (2, 4) & 24 \\ (4, 4) & 44 \\ (7, 4) & 74 \end{bmatrix}$$
In its full form, \( A \) looks like this:

```matlab
full(A)
```

```
ans =
    11     0    13     0
     0    22     0    24
     0     0    33     0
    41     0     0    44
     0    52     0     0
     0     0    63     0
     0     0     0    74
```

`spdiags` can also extract diagonal elements from a sparse matrix, or replace matrix diagonal elements with new values. Type `help spdiags` for details.

**Importing Sparse Matrices from Outside MATLAB**

You can import sparse matrices from computations outside MATLAB. Use the `spconvert` function in conjunction with the `load` command to import ASCII files containing lists of indices and nonzero elements. For example, consider a three-column text file `T.dat` whose first column is a list of row indices, second column is a list of column indices, and third column is a list of nonzero values. These statements load `T.dat` into MATLAB and convert it into a sparse matrix `S`:

```matlab
load T.dat
S = spconvert(T)
```

The `save` and `load` commands can also process sparse matrices stored as binary data in MAT-files. Finally, a Fortran utility routine `hbo2mat` is available to convert a file containing a sparse matrix in the Harwell-Boeing format into a MAT-file that `load` can process. The Harwell-Boeing data is available through anonymous ftp or the World Wide Web from `ftp.mathworks.com` in the directory `pub/mathworks/toolbox/matlab/sparfun`. 
Viewing Sparse Matrices

MATLAB provides a number of functions that let you get quantitative or graphical information about sparse matrices.

General Storage Information
The `whos` command provides high-level information about matrix storage, including size and storage class. For example, this `whos` listing shows information about sparse and full versions of the same matrix:

```
whos
Name           Size         Bytes  Class
M_full      1100x1100     9680000  double array
M_sparse    1100x1100        4404  sparse array

Grand total is 1210000 elements using 9684404 bytes
```

Notice that the number of bytes used is much less in the sparse case, because zero-valued elements are not stored. In this case, the density of the sparse matrix is 4404/9680000, or approximately .00045%.

Information About Nonzero Elements
There are several commands that provide high-level information about the nonzero elements of a sparse matrix:

- `nnz` returns the number of nonzero elements in a sparse matrix.
- `nonzeros` returns a column vector containing all the nonzero elements of a sparse matrix.
- `nzmax` returns the amount of storage space allocated for the nonzero entries of a sparse matrix.
To try some of these, load the supplied sparse matrix `west0479`, one of the Harwell-Boeing collection:

```matlab
load west0479
whos
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>west0479</td>
<td>479x479</td>
<td>24576</td>
<td>sparse array</td>
</tr>
</tbody>
</table>

This matrix models an eight-stage chemical distillation column.

Try these commands:

```matlab
nnz(west0479)
ans =
1887
```

```matlab
format short e
west0479
```

```matlab
west0479 =

(25, 1) 1.0000e+00
(31, 1) -3.7648e-02
(87, 1) -3.4424e-01
(26, 2) 1.0000e+00
(31, 2) -2.4523e-02
(88, 2) -3.7371e-01
(27, 3) 1.0000e+00
(31, 3) -3.6613e-02
(89, 3) -8.3694e-01
(28, 4) 1.3000e+02
```

...
nonzeros(west0479);

ans =

  1. 0000e+00
-3. 7648e-02
-3. 4424e-01
  1. 0000e+00
-2. 4523e-02
-3. 7371e-01
  1. 0000e+00
-3. 6613e-02
-8. 3694e-01
  1. 3000e+02
  .
  .
  .

**NOTE** Use Ctrl-C to stop the nonzeros listing at any time.

Note that initially `nnz` has the same value as `nzmax` by default. That is, the number of nonzero elements is equivalent to the number of storage locations allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of `nnz`, but not that of `nzmax`.

You can add as many nonzero elements to the matrix as desired, however; you are not constrained by the original value of `nzmax`.

**Viewing Sparse Matrices Graphically**

It is often useful to use a graphical format to view the distribution of the nonzero elements within a sparse matrix. MATLAB's `spy` function produces a template view of the sparsity structure, where each point on the graph represents the location of a nonzero array element.
The find Function and Sparse Matrices

For any matrix, full or sparse, the `find` function returns the indices and values of nonzero elements. Its syntax is:

```
[i, j, s] = find(S)
```

`find` returns the row indices of nonzero values in vector `i`, the column indices in vector `j`, and the nonzero values themselves in the vector `s`. The example below uses `find` to locate the indices and values of the nonzeros in a sparse matrix. The `sparse` function uses the `find` output, together with the size of the matrix, to recreate the matrix.

```
[i, j, s] = find(S)
[m n] = size(S)
S = sparse(i, j, s, m, n)
```
Example: Adjacency Matrices and Graphs

The formal mathematical definition of a graph is a set of points, or nodes, with specified connections between them. An economic model, for example, is a graph with different industries as the nodes and direct economic ties as the connections. The computer software industry is connected to the computer hardware industry, which, in turn, is connected to the semiconductor industry, and so on.

This definition of a graph lends itself to matrix representation. The adjacency matrix of an undirected graph is a matrix whose \((i,j)\)-th and \((j,i)\)-th entries are 1 if node \(i\) is connected to node \(j\), and 0 otherwise. For example, the adjacency matrix for a diamond-shaped graph looks like:

\[
A = \begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
\end{bmatrix}
\]

Since most graphs have relatively few connections per node, most adjacency matrices are sparse. The actual locations of the nonzero elements depend on how the nodes are numbered. A change in the numbering leads to permutation of the rows and columns of the adjacency matrix, which can have a significant effect on both the time and storage requirements for sparse matrix computations.

Graphing Using Adjacency Matrices

MATLAB's `gplot` function creates a graph based on an adjacency matrix and a related array of coordinates. To try `gplot`, create the adjacency matrix shown above by entering

\[
A = [0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0];
\]
The columns of `gplot`’s coordinate array contain the Cartesian coordinates for the corresponding node. For the diamond example, create the array by entering

```
xy = [1 3; 2 1; 3 3; 2 5];
```

This places the first node at location (1,3), the second at location (2,1), the third at location (3,3), and the fourth at location (2,5). To view the resulting graph, enter

```
gplot(A, xy)
```

**The Bucky Ball**

One interesting construction for graph analysis is the Bucky ball. This is composed of 60 points distributed on the surface of a sphere in such a way that the distance from any point to its nearest neighbors is the same for all the points. Each point has exactly three neighbors. The Bucky ball models four different physical objects:

- The geodesic dome popularized by Buckminster Fuller.
- The C\textsubscript{60} molecule, a form of pure carbon with 60 atoms in a nearly spherical configuration.
- In geometry, the truncated icosahedron.
- In sports, the seams in a soccer ball.

The Bucky ball adjacency matrix is a 60-by-60 symmetric matrix \( B \). \( B \) has three nonzero elements in each row and column, for a total of 180 nonzero values. This matrix has important applications related to the physical objects listed earlier. For example, the eigenvalues of \( B \) are involved in studying the chemical properties of C\textsubscript{60}.

To obtain the Bucky ball adjacency matrix, enter

```
B = bucky;
```

At order 60, and with a density of 5%, this matrix does not require sparse techniques, but it does provide an interesting example.

You can also obtain the coordinates of the Bucky ball graph using

```
[B, v] = bucky;
```
This statement generates \( v \), a list of \( xyz \)-coordinates of the 60 points in 3-space equidistributed on the unit sphere. The function \( \text{gplot} \) uses these points to plot the Bucky ball graph.

\[
\text{gplot}(B, v) \\
\text{axis equal}
\]

It is not obvious how to number the nodes in the Bucky ball so that the resulting adjacency matrix reflects the spherical and combinatorial symmetries of the graph. The numbering used by \texttt{bucky.m} is based on the pentagons inherent in the ball's structure.

The vertices of one pentagon are numbered 1 through 5, the vertices of an adjacent pentagon are numbered 6 through 10, and so on. The picture on the following page shows the numbering of half of the nodes (one hemisphere); the numbering of the other hemisphere is obtained by a reflection about the
equator. Use `gplot` to produce a graph showing half the nodes. You can add the node numbers using a `for` loop.

```matlab
k = 1:30;
gplot(B(k,k),v);
axis square
for j = 1:30, text(v(j,1),v(j,2), int2str(j)); end
```
To view a template of the nonzero locations in the Bucky ball’s adjacency matrix, use the `spy` function:

```matlab
spy(B)
```

The node numbering that this model uses generates a spy plot with twelve groups of five elements, corresponding to the twelve pentagons in the structure. Each node is connected to two other nodes within its pentagon and one node in some other pentagon. Since the nodes within each pentagon have consecutive numbers, most of the elements in the first super- and sub-diagonals of $B$ are nonzero. In addition, the symmetry of the numbering about the equator is apparent in the symmetry of the spy plot about the antidiagonal.
Graphs and Characteristics of Sparse Matrices
Spy plots of the matrix powers of B illustrate two important concepts related to sparse matrix operations, fill-in and distance. Spy plots help illustrate these concepts.

\[
\text{spy}(B^2) \\
\text{spy}(B^3) \\
\text{spy}(B^4) \\
\text{spy}(B^8)
\]

Fill-in is generated by operations like matrix multiplication. The product of two or more matrices usually has more nonzero entries than the individual terms, and so requires more storage. As \( p \) increases, \( B^p \) fills in and \( \text{spy}(B^p) \) gets more dense.
The distance between two nodes in a graph is the number of steps on the graph necessary to get from one node to the other. The spy plot of the $p$-th power of $B$ shows the nodes that are a distance $p$ apart. As $p$ increases, it is possible to get to more and more nodes in $p$ steps. For the Bucky ball, $B^8$ is almost completely full. Only the antidiagonal is zero, indicating that it is possible to get from any node to any other node, except the one directly opposite it on the sphere, in eight steps.

**An Airflow Model**

A calculation performed at NASA's Research Institute for Applications of Computer Science involves modeling the flow over an airplane wing with two trailing flaps.

In a two-dimensional model, a triangular grid surrounds a cross section of the wing and flaps. The partial differential equations are nonlinear and involve several unknowns, including hydrodynamic pressure and two components of velocity. Each step of the nonlinear iteration requires the solution of a sparse linear system of equations. Since both the connectivity and the geometric
location of the grid points are known, the `plot` function can produce the graph shown above.

In this example, there are 4253 grid points, each of which is connected to between 3 and 9 others, for a total of 28831 nonzeros in the matrix, and a density equal to 0.0016. This spy plot shows that the node numbering yields a definite band structure.

The Laplacian of the mesh.
Sparse Matrix Operations

Most of MATLAB’s standard mathematical functions work on sparse matrices just as they do on full matrices. In addition, MATLAB provides a number of functions that perform operations specific to sparse matrices. This section discusses:

- Computational Considerations
- Standard Mathematical Operations
- Permutation and Reordering
- Factorization
- Simultaneous Linear Equations
- Eigenvalues and Singular Values

Computational Considerations

The computational complexity of sparse operations is proportional to \( \text{nnz} \), the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size \( m \) and column size \( n \) of the matrix, but is independent of the product \( m \times n \), the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities. This is the “time is proportional to flops” rule.

Standard Mathematical Operations

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or vector always produce output in full storage format. For example, the size function always returns a full vector, whether its input is full or sparse.
- Functions that accept scalars or vectors and return matrices, such as zeros, ones, rand, and eye, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of zeros(m,n) is
simply \texttt{sparse(m,n)}. The sparse analogs of \texttt{rand} and \texttt{eye} are \texttt{sprand} and \texttt{speye}, respectively. There is no sparse analog for the function \texttt{ones}.

- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If \( S \) is a sparse matrix, then \texttt{chol(S)} is also a sparse matrix, and \texttt{diag(S)} is a sparse vector. Columnwise functions such as \texttt{max} and \texttt{sum} also return sparse vectors, even though these vectors may be entirely nonzero. Important exceptions to this rule are the \texttt{sparse} and \texttt{full} functions.

- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If \( S \) is sparse and \( F \) is full, then \( S+F, S*F, \) and \( F\backslash S \) are full, while \( S.*F \) and \( S\&F \) are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.

- Matrix concatenation using either the \texttt{cat} function or square brackets produces sparse results for mixed operands.

- Submatrix indexing on the right side of an assignment preserves the storage format of the operand. \( T = S(i,j) \) produces a sparse result if \( S \) is sparse whether \( i \) and \( j \) are scalars or vectors. Submatrix indexing on the left, as in \( T(i,j) = S \), does not change the storage format of the matrix on the left.

**Permutation and Reordering**

A permutation of the rows and columns of a sparse matrix \( S \) can be represented in two ways:

- A permutation matrix \( P \) acts on the rows of \( S \) as \( P\ast S \) or on the columns as \( S\ast P' \).

- A permutation vector \( p \), which is a full vector containing a permutation of \( 1:n \), acts on the rows of \( S \) as \( S(p,:) \), or on the columns as \( S(:,p) \).

For example, the statements

\[
\begin{align*}
\text{p} & = [1 \ 3 \ 4 \ 2 \ 5] \\
\text{l} & = \text{eye}(5, 5) \\
\text{P} & = \text{l}(\text{p}, :) \\
\text{e} & = \text{ones}(4, 1) \\
\text{S} & = \text{diag}(11:11:55) + \text{diag}(\text{e}, 1) + \text{diag}(\text{e}, -1)
\end{align*}
\]
produce

\[ p = \]
\[
1 \quad 3 \quad 4 \quad 2 \quad 5
\]

\[ P = \]
\[
\begin{matrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{matrix}
\]

\[ S = \]
\[
\begin{matrix}
11 & 1 & 0 & 0 & 0 \\
1 & 22 & 1 & 0 & 0 \\
0 & 1 & 33 & 1 & 0 \\
0 & 0 & 1 & 44 & 1 \\
0 & 0 & 0 & 1 & 55 \\
\end{matrix}
\]

You can now try some permutations using the permutation vector \( p \) and the permutation matrix \( P \). For example, the statements \( S(p,:) \) and \( P \times S \) produce

\[ \text{ans} = \]
\[
\begin{matrix}
11 & 1 & 0 & 0 & 0 \\
0 & 1 & 33 & 1 & 0 \\
0 & 0 & 1 & 44 & 1 \\
1 & 22 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 55 \\
\end{matrix}
\]

Similarly, \( S(:,p) \) and \( S \times P' \) produce

\[ \text{ans} = \]
\[
\begin{matrix}
11 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 22 & 0 \\
0 & 33 & 1 & 1 & 0 \\
0 & 1 & 44 & 0 & 1 \\
0 & 0 & 1 & 0 & 55 \\
\end{matrix}
\]
If \( P \) is a sparse matrix, then both representations use storage proportional to \( n \) and you can apply either to \( S \) in time proportional to \( \text{nnz}(S) \). The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return full row vectors with the exception of the pivoting permutation in LU (triangular) factorization, which returns a matrix compatible with earlier versions of MATLAB.

To convert between the two representations, let \( I = \text{sp.eye}(n) \) be an identity matrix of the appropriate size. Then,

\[
P = I(p,:)
\]
\[
P' = I(:,p).
\]
\[
p = (1:n) \cdot P'
\]
\[
p = (P \cdot (1:n)')'
\]

The inverse of \( P \) is simply \( R = P' \). You can compute the inverse of \( p \) with \( r(p) = 1:n \).

\[
r(p) = 1:5
\]

\[
r =
\]
\[
1 \quad 4 \quad 2 \quad 3 \quad 5
\]

**Reordering for Sparsity**

Reordering the columns of a matrix can often make its Cholesky, LU, or QR factors sparser. The simplest such reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The function \( p = \text{col.perm}(S) \) computes this column-count permutation. The \texttt{col.perm} M-file has only a single line:

\[
[\text{ignore}, p] = \text{sort}\left(\text{full}\left(\text{sum}(\text{spones}(S))\right)\right);
\]
This line performs these steps:

1. The inner call to `spones` creates a sparse matrix with ones at the location of every nonzero element in S.

2. The `sum` function sums down the columns of the matrix, producing a vector that contains the count of nonzeros in each column.

3. `full` converts this vector to full storage format.

4. `sort` sorts the values in ascending order. The second output argument from `sort` is the permutation that sorts this vector.

**Reordering to Reduce Bandwidth**

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix. It is not guaranteed to find the smallest possible bandwidth, but it usually does. The function `symrcm(A)` actually operates on the nonzero structure of the symmetric matrix $A + A'$, but the result is also useful for asymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense “long and thin.”

**Minimum Degree Ordering**

The degree of a node in a graph is the number of connections to that node, which is the same as the number of nonzero elements in the corresponding row of the adjacency matrix. The minimum degree algorithm generates an ordering based on how these degrees are altered during Gaussian elimination. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee. MATLAB has two versions, `symmd` for symmetric matrices and `colmmd` for nonsymmetric matrices. You can change various parameters associated with details of the algorithm using the `spparms` function.


**Factorization**

This section discusses four important factorization techniques for sparse matrices:

- LU, or triangular, factorization
- Cholesky factorization
- QR, or orthogonal factorization
- Incomplete factorizations

**LU Factorization**

If \( S \) is a sparse matrix, the statement below returns three sparse matrices \( L \), \( U \), and \( P \) such that \( P \times S = L \times U \).

\[
[L, U, P] = lu(S)
\]

\( lu \) obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix \( P \) has only \( n \) nonzero elements. As with dense matrices, the statement \([ L, U \]) = lu(S)\) returns a permuted unit lower triangular matrix and an upper triangular matrix whose product is \( S \). By itself, \( lu(S) \) returns \( L \) and \( U \) in a single matrix without the pivot information.

The sparse LU factorization does not pivot for sparsity, but it does pivot for numerical stability. In fact, both the sparse factorization (line 1) and the full factorization (line 2) below produce the same \( L \) and \( U \), even though the time and storage requirements might differ greatly:

\[
[L, U] = lu(S) \quad \text{% sparse factorization}
\]

\[
[L, U] = sparse(lu(full(S))) \quad \text{% full factorization}
\]

MATLAB automatically allocates the memory necessary to hold the sparse \( L \) and \( U \) factors during the factorization. MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data structures in advance.

**Reordering and factorization.** If you obtain a good column permutation \( p \) that reduces fill-in, perhaps from \( symrcm \) or \( colamd \), then computing \( lu(S(:, p)) \) will take less time and storage than computing \( lu(S) \). Two permutations are
the symmetric reverse Cuthill-McKee ordering and the symmetric minimum degree ordering:

\[ r = \text{symrcm}(B); \]
\[ m = \text{symmmd}(B); \]

The three spy plots produced by the lines below show the three adjacency matrices of the Bucky Ball graph with these three different numberings. The local, pentagon-based structure of the original numbering is not evident in the other three.

\[ \text{spy}(B) \]
\[ \text{spy}(B(r, r)) \]
\[ \text{spy}(B(m, m)) \]

The reverse Cuthill-McGee ordering, \( r \), reduces the bandwidth and concentrates all the nonzero elements near the diagonal. The minimum degree ordering, \( m \), produces a fractal-like structure with large blocks of zeros.

To see the fill-in generated in the LU factorization of the Bucky ball, use \( \text{speye}(n, n) \), the sparse identity matrix, to insert -3s on the diagonal of \( B \):

\[ B = B - 3*\text{speye}(n, n) \]

Since each row sum is now zero, this new \( B \) is actually singular, but it is still instructive to compute its LU factorization. When called with only one output argument, \( \text{lu} \) returns the two triangular factors, \( L \) and \( U \), in a single sparse matrix. The number of nonzeros in that matrix is a measure of the time and
storage required to solve linear systems involving B. Here are the nonzero counts for the three permutations being considered:

<table>
<thead>
<tr>
<th></th>
<th>l u( B)</th>
<th>1022</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse Cuthill-McKee</td>
<td>l u( B(r, r) )</td>
<td>968</td>
</tr>
<tr>
<td>Minimum degree</td>
<td>l u( B(m m) )</td>
<td>660</td>
</tr>
</tbody>
</table>

Even though this is a small example, the results are typical. The original numbering scheme leads to the most fill-in. The fill-in for the reverse Cuthill-McKee ordering is concentrated within the band, but it is almost as extensive as the first two orderings. For the minimum degree ordering, the relatively large blocks of zeros are preserved during the elimination and the amount of fill-in is significantly less than that generated by the other orderings. The spy plots below reflect the characteristics of each reordering.

**Cholesky Factorization**

If S is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix R so that R' * R = S.

\[
R = \text{chol}(S)
\]

\text{chol} does not automatically pivot for sparsity, but you can compute minimum degree and profile limiting permutations for use with \text{chol}(S(p, p)).
Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, it is possible to do a quick calculation of the amount of memory required and allocate all the memory at the start of the factorization.

**QR Factorization**

MATLAB will compute the complete QR factorization of a sparse matrix \( S \) with

\[
[Q, R] = qr(S)
\]

but this is usually impractical. The orthogonal matrix \( Q \) often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as “the Q-less QR factorization,” is available.

With one sparse input argument and one output argument

\[
R = qr(S)
\]

returns just the upper triangular portion of the QR factorization. The matrix \( R \) provides a Cholesky factorization for the matrix associated with the normal equations,

\[
R' * R = S' * S
\]

However, the loss of numerical information inherent in the computation of \( S' * S \) is avoided.

With two input arguments having the same number of rows, and two output arguments, the statement

\[
[C, R] = qr(S, B)
\]

applies the orthogonal transformations to \( B \), producing \( C = Q' * B \) without computing \( Q \).

The Q-less QR factorization allows the solution of sparse least squares problems

\[
\text{minimize} ||Ax - b||
\]

with two steps

\[
[C, R] = qr(A, b)
\]
\[
x = R \setminus c
\]
If \( A \) is sparse, but not square, MATLAB uses these steps for the linear equation solving backslash operator
\[
x = A \backslash b
\]

Or, you can do the factorization yourself and examine \( R \) for rank deficiency.

It is also possible to solve a sequence of least squares linear systems with different right hand sides, \( b \), that are not necessarily known when \( R = qr(A) \) is computed. The approach solves the “seminormal equations”
\[
R' R x = A' b
\]
with
\[
x = R \backslash (R' \backslash (A' b))
\]
and then employs one step of iterative refinement to reduce roundoff error
\[
r = b - A x
\]
\[
e = R \backslash (R' \backslash (A' r))
\]
\[
x = x + e
\]

**Incomplete Factorizations**

The `luinc` and `cholinc` functions provide approximate, incomplete factorizations, which are useful as preconditioners for sparse iterative methods.

The `luinc` function produces two different kinds of incomplete LU factorizations, one involving a drop tolerance and one involving fill-in level. If \( A \) is a sparse matrix, and \( tol \) is a small tolerance, then
\[
[L, U] = luinc(A, tol)
\]
computes an approximate LU factorization where all elements less than \( tol \) times the norm of the relevant column are set to zero. Alternatively,
\[
[L, U] = luinc(A, '0')
\]
computes an approximate LU factorization where the sparsity pattern of \( L+U \) is a permutation of the sparsity pattern of \( A \).
For example

```matlab
load west0479
A = west0479;
nnz(A)
nnz(lu(A))
nnz(lui nc(A, 1e-6))
nnz(lui nc(A, '0'))
```

shows that \( A \) has 1887 nonzeros, its complete LU factorization has 16777 nonzeros, its incomplete LU factorization with a drop tolerance of 1e-6 has 10311 nonzeros, and its \( \text{lu('0')} \) factorization has 1886 nonzeros.

The \( \text{lui nc} \) function has a few other options. See the online help for details.

The \( \text{cholinc} \) function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the online help for more information.

**Simultaneous Linear Equations**

Systems of simultaneous linear equations can be solved by two different classes of methods:

- **Direct methods.** These are usually variants of Gaussian elimination and are often expressed as matrix factorizations such as LU or Cholesky factorization. The algorithms involve access to the individual matrix elements.

- **Iterative methods.** Only an approximate solution is produced after a finite number of steps. The coefficient matrix is involved only indirectly, through a matrix-vector product or as the result of an abstract linear operator.

**Direct Methods**

Direct methods are usually faster and more generally applicable, if there is enough storage available to carry them out. Iterative methods are usually applicable to restricted cases of equations and depend upon properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of MATLAB and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB M-files and may make use of the direct solution of subproblems or preconditioners.
The usual way to access direct methods in MATLAB is not through the \texttt{lu} or \texttt{chol} functions, but rather with the matrix division operators \texttt{/} and \texttt{\textbackslash}. If \( A \) is square, the result of \( X = A\backslash B \) is the solution to the linear system \( A^*X = B \). If \( A \) is not square, then a least squares solution is computed.

If \( A \) is a square, full, or sparse matrix, then \( A\backslash B \) has the same storage class as \( B \). Its computation involves a choice among several algorithms.

- If \( A \) is triangular, perform a triangular solve for each column of \( B \).
- If \( A \) is a permutation of a triangular matrix, permute it and perform a sparse triangular solve for each column of \( B \).
- If \( A \) is symmetric or Hermitian and has positive real diagonal elements, find a symmetric minimum degree order \( p \) and attempt to compute the Cholesky factorization of \( A(p, p) \). If successful, finish with two sparse triangular solves for each column of \( B \).
- Otherwise (if \( A \) is not triangular, or is not Hermitian with positive diagonal, or if Cholesky factorization fails), find a column minimum degree order \( p \). Compute the LU factorization with partial pivoting of \( A(:, p) \), and perform two triangular solves for each column of \( B \).

For a square matrix, MATLAB tries these possibilities in order of increasing cost. The tests for triangularity and symmetry are relatively fast and, if successful, allow for faster computation and more efficient memory usage than the general purpose method.

For example, consider the sequence below.

\[
[ L, U ] = \text{lu}( A );
\]

\[
y = L\backslash b;
\]

\[
x = U\backslash y;
\]

In this case, MATLAB uses triangular solves for both matrix divisions, since \( L \) is a permutation of a triangular matrix and \( U \) is triangular.

Use the function \texttt{spparms} to turn off the minimum degree preordering if a better preorder is known for a particular matrix.
Iterative Methods
Six functions are available that implement iterative methods for sparse systems of simultaneous linear systems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bi cg</td>
<td>Biconjugate gradient.</td>
</tr>
<tr>
<td>bi cgst ab</td>
<td>Biconjugate gradient stabilized.</td>
</tr>
<tr>
<td>cgs</td>
<td>Conjugate gradient squared.</td>
</tr>
<tr>
<td>gmres</td>
<td>Generalized minimum residual.</td>
</tr>
<tr>
<td>pcg</td>
<td>Preconditioned conjugate gradient.</td>
</tr>
<tr>
<td>qmr</td>
<td>Quasiminimal residual.</td>
</tr>
</tbody>
</table>

All six methods are designed to solve $Ax = b$. The preconditioned conjugate gradient method, pcg, is restricted to symmetric, positive definite matrix $A$. The other five can handle nonsymmetric, square matrices.

All six methods can make use of left and right preconditioners. The linear system

$$Ax = b$$

is replaced by the equivalent system

$$M_1^{-1}A^{-1}M_2^{-1}M_2x = M_1^{-1}b$$

The preconditioners $M_1$ and $M_2$ are chosen to accelerate convergence of the iterative method. In many cases, the preconditioners occur naturally in the mathematical model. A partial differential equation with variable coefficients may be approximated by one with constant coefficients, for example. Incomplete matrix factorizations may be used in the absence of natural preconditioners.

The five-point finite difference approximation to Laplace’s equation on a square, two-dimensional domain provides an example. The following
statements use the preconditioned conjugate gradient method with an
incomplete Cholesky factorization as a preconditioner.

\[ A = \text{delsq}(\text{numgrid}('S', 50)); \]
\[ b = \text{ones(size(A, 1), 1)}; \]
\[ \text{tol} = 1. \times 10^{-3}; \]
\[ \text{maxit} = 10; \]
\[ R = \text{cholinc}(A, \text{tol}); \]
\[ [x, \text{flag}, \text{err}, \text{iter}, \text{res}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, R', R); \]

Only four iterations are required to achieve the prescribed accuracy.

Background information on these iterative methods and incomplete
factorizations is available in:

Saad, Yousef. \textit{Iterative Methods for Sparse Linear Equations}. PWS Publishing
Company: 1996.

Barrett, Richard et al. \textit{Templates for the Solution of Linear Systems: Building
Blocks for Iterative Methods}. Society for Industrial and Applied Mathematics:
1994.

\section*{Eigenvalues and Singular Values}

Two functions are available which compute a few specified eigenvalues or
singular values.

\begin{tabular}{|l|l|}
\hline
\textbf{Function} & \textbf{Description} \\
\hline
\texttt{eigs} & Few eigenvalues. \\
\texttt{svds} & Few singular values. \\
\hline
\end{tabular}

These functions are most frequently used with sparse matrices, but they can be
used with full matrices or even with linear operators defined by M-files.

The statement

\[ [V, \lambda \text{andbda}] = \text{eigs}(A, k, \text{sigma}) \]

finds the \( k \) eigenvalues and corresponding eigenvectors of the matrix \( A \) which
are nearest the “shift” \( \text{sigma} \). If \( \text{sigma} \) is omitted, the eigenvalues largest in
magnitude are found. If \( \text{sigma} \) is zero, the eigenvalues smallest in magnitude
are found. A second matrix, \( B \), may be included for the generalized eigenvalue problem

\[
A v = \lambda B v
\]

The statement

\[
[ U, S, V ] = \text{svds}( A, k )
\]

finds the \( k \) largest singular values of \( A \) and

\[
[ U, S, V ] = \text{svds}( A, k, 0 )
\]

finds the \( k \) smallest singular values.

For example, the statements

\[
L = \text{numgrid('L',65)};
A = \text{delsq}(L);
\]

set up the five-point Laplacian difference operator on a 65-by-65 grid in an L-shaped, two-dimensional domain. The statements

\[
\text{size}(A)
\text{nnz}(A)
\]

show that \( A \) is a matrix of order 2945 with 14,473 nonzero elements.

The statement

\[
[ v, d ] = \text{eigs}( A, 1, 0 );
\]

computes the smallest eigenvalue and eigenvector. Finally,

\[
L(L>0) = \text{full}(v(L(L>0)));
x = -1:1/32:1;
\text{contour}(x,x,L,15);
\text{axis square}
\]
distributes the components of the eigenvector over the appropriate grid points and produces a contour plot of the result.

The numerical techniques used in `eigs` and `svds` are described in a paper by D. C. Sorensen, *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations*. A copy of the paper is available through the MATLAB Help Desk.
# M-File Programming

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB Programming: A Quick Start</td>
<td>10-2</td>
</tr>
<tr>
<td>Scripts</td>
<td>10-5</td>
</tr>
<tr>
<td>Local and Global Variables</td>
<td>10-16</td>
</tr>
<tr>
<td>Data Types</td>
<td>10-19</td>
</tr>
<tr>
<td>Operators</td>
<td>10-22</td>
</tr>
<tr>
<td>Flow Control</td>
<td>10-29</td>
</tr>
<tr>
<td>Subfunctions</td>
<td>10-35</td>
</tr>
<tr>
<td>Indexing and Subscripting</td>
<td>10-37</td>
</tr>
<tr>
<td>String Evaluation</td>
<td>10-43</td>
</tr>
<tr>
<td>Command/Function Duality</td>
<td>10-45</td>
</tr>
<tr>
<td>Empty Matrices</td>
<td>10-46</td>
</tr>
<tr>
<td>Errors and Warnings</td>
<td>10-48</td>
</tr>
<tr>
<td>Times and Dates</td>
<td>10-51</td>
</tr>
<tr>
<td>Obtaining User Input</td>
<td>10-57</td>
</tr>
<tr>
<td>Shell Escape Functions</td>
<td>10-58</td>
</tr>
<tr>
<td>Optimizing the Performance of MATLAB Code</td>
<td>10-59</td>
</tr>
</tbody>
</table>
MATLAB Programming: A Quick Start

Files that contain MATLAB language code are called M-files. M-files can be functions that accept arguments and produce output, or they can be scripts that execute a series of MATLAB statements. You create M-files using a text editor, then use them as you would any other MATLAB function or command. The process looks like this:

1. Create an M-file using a text editor.
   ```matlab
   function c = myfile(a, b)
   c = sqrt((a.^2)+(b.^2))
   end
   ```

2. Call the M-file from the command line, or from within another M-file.
   ```matlab
   >> a = 7.5
   >> b = 3.342
   >> c = myfile(a, b)
   c = 8.2109
   ```

Kinds of M-Files
There are two kinds of M-files:

<table>
<thead>
<tr>
<th>Script M-Files</th>
<th>Function M-Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Do not accept input arguments or return output arguments</td>
<td>• Can accept input arguments and return output arguments</td>
</tr>
<tr>
<td>• Operate on data in the workspace</td>
<td>• Internal variables are local to the function by default</td>
</tr>
<tr>
<td>• Useful for automating a series of steps you need to perform many times</td>
<td>• Useful for extending the MATLAB language for your application</td>
</tr>
</tbody>
</table>
What's in an M-File?

This section shows you the basic parts of a function M-file, so you can familiarize yourself with MATLAB programming and get started with some examples.

```matlab
function f = fact(n)
    % FACT Factorial.
    % FACT(N) returns the factorial of N, usually denoted by N!.
    % Put simply, FACT(N) is PROD(1:N).
    f = prod(1:n);
end
```

This function has some elements that are common to all MATLAB functions:

- A function definition line. This line defines the function name, and the number and order of input and output arguments.
- A H1 line. H1 stands for “help 1” line. MATLAB displays the H1 line for a function when you use `lookfor` or request help on an entire directory.
- Help text. MATLAB displays the help text entry together with the H1 line when you request help on a specific function.
- The function body. This part of the function contains code that performs the actual computations and assigns values to any output arguments.

The “Functions” section coming up provides more detail on each of these parts of a MATLAB function.

Creating M-Files: Accessing Text Editors

M-files are ordinary text files that you create using a text editor. MATLAB provides a built-in editor for the PC and Macintosh environments, although you can use any ASCII text editor you like. MATLAB does not provide a default editor for UNIX systems, so use any editor you’re comfortable with.

To open the editor on the PC and Macintosh
From the File menu choose New then M-File.

Another way to edit an M-file is from the MATLAB command line using the `edit` command. For example,

```matlab
edit poof
```
opens the editor on the file poof.m. Omitting a filename opens the editor on an untitled file.

You can create the fact function shown on the previous page 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've created this file, here are some things you can do

- 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
  ```
Scripts

Scripts are the simplest kind of M-file – 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. Scripts 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.

Simple Script Example

These statements calculate $\rho$ for several trigonometric functions of $\theta$, then create a series of polar plots.

```matlab
% An M-file script to produce "flower petal" plots
theta = -pi:0.01:pi;
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 i = 1:4
    polar(theta, rho(i,:))
    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 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.

Functions

Functions are M-files that accept input arguments and return output arguments. They operate on variables within their own workspace, separate from the workspace you access at the MATLAB command line prompt.
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.
% Non-vector 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
```

If you would like, 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
```

The Anatomy of a Function
A function M-file consists of

- A function definition line
- A H1 line
- Function help text
- The function body
- Comments
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 average function is

```matlab
function y = average(x)
```

All MATLAB functions have a function definition line that follows this pattern. If the function has more than one output argument, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses. Use commas to separate multiple input or output arguments. Here's a more complicated example:

```matlab
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```matlab
function printresults(x)
```

or use empty square brackets

```matlab
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.

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

```matlab
% AVERAGE Mean of vector elements.
```
This is the first line of text that appears when a user types `help function_name` at the MATLAB prompt. Further, the `lookfor` command 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 text on one or more comment lines, beginning with the line immediately following the H1 line. The help text for the `average` function is

```matlab
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Non-vector input results in an error.
```

When you type `help function_name`, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line. The help system ignores any comment lines that appear after this help block.

For example, typing `help sin` results in

```
SIN     Sine.
SIN(X) is the sine of the elements of X.
```

**Function 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:

\[
\begin{align*}
[m, n] &= \text{size}(x); \\
\text{if} \ (\neg ((m == 1) \ | \ (n == 1)) \ | \ (m == 1 \ & \ n == 1)) \\
\text{error}('\text{Input must be a vector}') \\
\text{end} \\
y &= \text{sum}(x) / \text{length}(x);
\end{align*}
\]

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,

\[
\begin{align*}
\% \text{Add up all the vector elements.} \\
y &= \text{sum}(x) \quad \% \text{Use the sum function.}
\end{align*}
\]

The first comment line immediately following the function definition line is
considered the H1 line for the function. The H1 line and any comment lines
immediately following it constitute the online help entry for the file.

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.

Function Names
MATLAB function names have the same constraints as variable names.
MATLAB uses the first 31 characters of names. Function names must begin
with a letter; the remaining characters can be any combination of letters,
numbers, and underscores. Some operating systems may restrict function
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 filename
wins and the internal name is ignored.
Thus, while the function name specified on the function definition line does not have to be the same as the filename, we strongly recommend that you use the same name for both.

How Functions Work
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.

Function Name Resolution
When MATLAB comes upon a new name, it resolves it into a specific function by following these steps:

1. Checks to see if the name is a variable.

2. Checks to see if the name is a subfunction, a MATLAB function that resides in the same M-file as the calling function. Subfunctions are discussed on page 10-35.

3. Checks to see if the name is a private function, a MATLAB function that resides in a private directory, a directory accessible only to M-files in the directory immediately above it. Private directories are discussed on page 10-36.

4. Checks to see if the name is a function on the MATLAB search path. MATLAB uses the first file it encounters with the specified name.

If you duplicate function names, MATLAB executes the one found first using the above rules. It is also possible to overload function names. This uses additional dispatching rules and is discussed in Chapter 14, “Classes and Objects.”

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` command, or until you quit MATLAB. Variants of the `clear` command that you can use to clear functions from memory include:

- `clear function_name` Remove specified function from workspace.
- `clear functions` Remove all compiled M-functions.
- `clear all` Remove all variables and functions

**Creating P-Code Files**

You can save a preparsed version of a function or script for later MATLAB sessions using the `pcode` command. 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` command 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.

Another situation for `pcode` is when, for proprietary reasons, you wish to hide algorithms you've created in your M-file.

**How MATLAB Passes Function Arguments**

From the programmer's perspective, MATLAB appears to pass all function arguments by value. Actually, however, MATLAB passes by value only those arguments that a function modifies. If a function does not alter an argument but simply uses it in a computation, MATLAB passes the argument by reference to optimize memory use.

**Function Workspaces**

Each M-file function has an area of memory, separate from MATLAB's base workspace, in which it operates. This area is called the function workspace, with each function having 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.

**Checking the Number of Function Arguments**

The `nargin` and `nargout` functions let you 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,

```matlab
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’s a more advanced example that finds the first token in a character string. A token is a set of characters delimited by whitespace or some other character. Given one input, the function assumes a default delimiter of whitespace; 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)
  if nargin < 1, error('Not enough input arguments.'); end
  token = []; remainder = [];
  len = length(string);
  if len == 0
    return
  end
  if (nargin == 1)
    delimiters = [9:13 32]; % White space characters
  end
  i = 1;
  while (any(string(i) == delimiters))
    i = i + 1;
    if (i > len), return, end
  end
  start = i;
  while (~any(string(i) == delimiters))
    i = i + 1;
    if (i > len), break, end
  end
  finish = i - 1;
  token = string(start:finish);
  if (nargout == 2)
    remainder = string(finish + 1:end);
  end

NOTE: strtok is a MATLAB M-file in the strfun directory.

Note that 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. MATLAB packs all of the specified input or output 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.

Here's an example function that accepts any number of two-element vectors and draws a line to connect them:

```matlab
function testvar(varargin)
    for i = 1:length(varargin)
        x(i) = varargin{i}(1);
        y(i) = varargin{i}(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)
end
```

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(i) = varargin{i}(2);`

Cell array indexing has two subscript components:

- The cell indexing expression, in curly braces
- The contents indexing expression(s), in parentheses

In the code above, the indexing expression `{i}` accesses the i'th cell of `varargin`. The expression `(2)` represents the second element of the cell contents.
Packing varargout Contents
When allowing any 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 \([ x_i \ y_i] \) vectors that you can pass into the \texttt{testvar} function on the previous page:

\[
\text{function } [\text{varargout}]=\text{testvar}(\text{arrayin})
\]
\[
\text{for } i=1:\text{nargout}
\]
\[
\text{varargout} \{i\} = \text{arrayin}(i,:)
\]
\[
\text{end}
\]

The assignment statement inside the \texttt{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 “Structures and Cell Arrays” in Chapter 13.

Here’s how to call \texttt{testvar}.

\[
a=[1\ 2\ 3\ 4\ 5;6\ 7\ 8\ 9\ 0]';
\[\ p1, p2, p3, p4, p5\] = \text{testvar}(a)
\]

\texttt{varargin and varargout in Argument Lists}
\texttt{varargin} or \texttt{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 \texttt{varargin} and \texttt{varargout}:

\[
\text{function } [\text{out1}, \text{out2}]=\text{example1}(a, b, \text{varargin})
\]
\[
\text{function } [i, j, \text{varargout}]=\text{example2}(x1, y1, x2, y2, \text{flag})
\]
Local and Global 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. Before assigning one variable to another, however, 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.
- MATLAB variable names consist of a letter followed by any number of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so A and a are not the same variable.
- MATLAB uses only the first 31 characters of variable names.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, 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 you want to study the effect of the interaction coefficients, \( \alpha \) and \( \beta \), in the Lotka-Volterra predator-prey model

\[
\begin{align*}
\dot{y}_1 &= y_1 - \alpha y_1 y_2 \\
\dot{y}_2 &= -y_2 + \beta y_1 y_2
\end{align*}
\]

Create an M-file, `lotka.m`

```matlab
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)];
```

\[
\begin{align*}
\dot{y}_1 &= y_1 - \alpha y_1 y_2 \\
\dot{y}_2 &= -y_2 + \beta y_1 y_2
\end{align*}
\]
Then interactively enter the statements:

```matlab
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.

For your MATLAB application to work with global variables:

1. Declare the variable as global in every function that requires access to it. To enable the workspace to access the global variable, also declare it as global from the command line.
2. In each function, issue the global command before the first occurrence of the variable name. The top of the M-file is recommended.

MATLAB global variable names are typically longer and more descriptive than local variable names, and sometimes consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code and reduce the chance of accidentally redefining a global variable.

**Special Values**

Several functions return important special values that you can use in your M-files:

- **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.
- **realmax**: Largest floating-point number your computer can represent.
- **realmin**: Smallest floating-point number your computer can represent.
- **pi**: 3.1415926535897...
- **i, j**: Imaginary unit.
All of these special functions and constants reside in MATLAB's `elmat` directory, and provide online help. Here are several examples that use them in MATLAB expressions:

```matlab
x = 2*pi;
A = [3+2i 7-8i];
tol = 3*eps;
```

### Functions and Constants

- **`inf`**: Infinity. Calculations like `n/0`, where `n` is any nonzero numeric 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`.
- **`computer`**: Computer type.
- **`flops`**: Count of floating-point operations.
- **`version`**: MATLAB version string.
Data Types

There are six fundamental data types (classes) in MATLAB, each one a multidimensional array. The six classes are double, char, sparse, uint8, cell, and struct. The two-dimensional versions of these arrays are called matrices and are where MATLAB gets its name.

```
array
  | char  numeric  cell  struct
  |                  |
  | double          uint8
  | sparse
```

You will probably spend most of your time working with only two of these data types: the double precision matrix (double) and the character array (char) or string. This is because all computations are done in double-precision and most of the functions in MATLAB work with arrays of double-precision numbers or strings.

The other data types are for specialized situations like image processing (uint8), sparse matrices (sparse), and large scale programming (cell and struct).

You can't create variables with the types “numeric” or “array”. These virtual types serve only to group together types that share some common attributes.

The uint8 data type is for memory efficient storage only. You can apply basic operations such as subscripting and reshaping to these types of arrays but you can't perform any math with them. You must convert such arrays to double via the double function before doing any math operations.

Creating Your Own Type or Adding Methods for a Built-In Type

The tables show a seventh data type, the UserObject. The MATLAB language allows you to create your own data types and to work with them the same way you work with the built-in types. When you do so, you create a UserObject.
You overload methods for the built-in data types in exactly the same way you overload a method for an object. For example to define `sort` for a `uint8` array, create a method (`sort.m` or `sort.mex`) and place it into a `@uint8` directory within a directory on your path.

The table that follows describes the data types in more detail.

<table>
<thead>
<tr>
<th>Class</th>
<th>Example</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double</code></td>
<td><code>[ 1 2; 3 4]</code></td>
<td>5+6i</td>
<td>Double precision numeric array (this is the most common MATLAB variable type).</td>
</tr>
<tr>
<td><code>char</code></td>
<td><code>'Hello'</code></td>
<td></td>
<td>Character array (each character is 16 bits long). Also referred to as a string.</td>
</tr>
<tr>
<td><code>sparse</code></td>
<td>speye(5)</td>
<td></td>
<td>Sparse double precision matrix (2-D only). The sparse matrix stores matrices with only a few non-zero elements in a fraction of the space required for an equivalent full matrix. Sparse matrices invoke special methods especially tailored to solve sparse problems.</td>
</tr>
<tr>
<td><code>cell</code></td>
<td><code>{17 'Hello' eye(2)}</code></td>
<td></td>
<td>Cell array. Elements of cell arrays contain other arrays. Cell arrays collect related data and information of a dissimilar size together.</td>
</tr>
<tr>
<td><code>struct</code></td>
<td><code>a.day = 12; a.color = 'Red'; a.mat = magic(3);</code></td>
<td></td>
<td>Structure array. Structure arrays have field names. The fields contain other arrays. Like cell arrays, structures collect related data and information together.</td>
</tr>
<tr>
<td><code>uint8</code></td>
<td><code>uint8(magic(3))</code></td>
<td></td>
<td>Unsigned 8 bit integer array. The <code>uint8</code> array stores integers in the range from 0 to 255 in 1/8 the memory required for a double precision array. No mathematical operations are defined for <code>uint8</code> arrays.</td>
</tr>
<tr>
<td><code>UserObject</code></td>
<td><code>inline('sin(x)')</code></td>
<td></td>
<td>User-defined data type.</td>
</tr>
</tbody>
</table>
Reading the Diagram

The connecting lines in the diagram on page 10-19 are “isa” relationships. For instance, the sparse matrix type “is a” double and “is a” numeric, hence

```plaintext
isa(s, 'sparse')
isa(s, 'double')
isa(s, 'numeric')
```

all return 1 (true) when s is sparse. Note that `array` is at the top of the hierarchy. This shows that all data types in MATLAB are arrays.

Each type in the diagram supports certain functions and operators (or methods). The child types below a parent type also support the methods of the parent. Hence the double precision array supports all the methods of `array`, `numeric`, as well as `double`. Here are some of the methods that are supported for the types:

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>Multidimensional subscripting, reshape, size, length, ndims, concatenation <code>[a b]</code>, transpose, permute</td>
</tr>
<tr>
<td>cell</td>
<td>Cell array content subscripting via {}, comma separated list expansion</td>
</tr>
<tr>
<td>char</td>
<td>String functions (strcmp, lower), automatic conversion to double to support most of the double methods</td>
</tr>
<tr>
<td>double</td>
<td>Math operators (+,-,*, etc), logical operators(&gt;=, etc), matrix functions (eig, svd, etc.), mathematical functions (sin, cos, sum, prod, sort, etc.)</td>
</tr>
<tr>
<td>numeric</td>
<td>Find, complex numbers (real, imag), colon (1:10), scalar expansion</td>
</tr>
<tr>
<td>sparse</td>
<td>Sparse math (/, *, spl u, spchol, etc.)</td>
</tr>
<tr>
<td>struct</td>
<td>Structure field content access via .field or syntax, comma-separated list expansion</td>
</tr>
<tr>
<td>uint8</td>
<td>Storage only (more supported with the Image Processing Toolbox)</td>
</tr>
<tr>
<td>UserObject</td>
<td>User defined</td>
</tr>
</tbody>
</table>
Operators

MATLAB's operators fall into three categories.

- Arithmetic operators that perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power.
- Relational operators that compare operands quantitatively, using operators like "less than" and "not equal to."
- Logical operators that use the logical operators AND, OR, and NOT, with the result corresponding to TRUE or FALSE.

Arithmetic operators have the highest precedence, then relational operators, and then logical operators.

Arithmetic Operators

The precedence rules for the arithmetic operators are:

1. transpose (\texttt{.'}), power (\texttt{.^}), complex conjugate transpose (\texttt{'}), matrix power (\texttt{^})
2. unary plus (\texttt{+}), unary minus (\texttt{-})
3. multiplication (\texttt{.*}), right division (\texttt{./}), left division (\texttt{.ackslash}), matrix multiplication (\texttt{*}), matrix right division (\texttt{/}), matrix left division (\texttt{ackslash})
4. addition (\texttt{+}), subtraction (\texttt{-})
5. colon operator (\texttt{:})

Within each precedence level, operators have equal precedence and are evaluated from left to right.

The default precedence can be overridden using parentheses. For example, given two vectors

\begin{verbatim}
A = [3 9 5];
B = [2 1 5];
\end{verbatim}

the statements

\begin{verbatim}
C = A ./ B.^2
\end{verbatim}
and
\[ C = (A ./ B)^2 \]
result in different \( C \) arrays.

Expressions can also include values that you access through subscripts:
\[ b = \sqrt{A(2)} + 2 \times B(1) \]

\[ b = 7 \]

Except for some matrix operators, MATLAB’s 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.

**Relational Operators**

There are six relational operators:

- `<` Less than
- `<=` Less than or equal to
- `>` Greater than
- `>=` Greater than or equal to
- `==` Equal to
- `~=` Not equal to

MATLAB’s relational operators compare corresponding elements of arrays with equal dimensions. 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 the value 1. Locations where the relation is false receive the value 0.
Relational operators are useful for flow control. A relational expression often controls the execution of statements within an if, for, while, or switch construct.

Relational operators always operate element-by-element. For example,

```matlab
A = [2 7 6; 9 0 -1; 3 0.5 6];
B = [8 0.2 0; -3 2.5; 4 -1 7];
A < B
```

```
ans =

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

The resulting matrix shows where an element of A is less than the corresponding element of B.

The relational operators have the second highest precedence when MATLAB evaluates expressions, with the arithmetic operators being first and the logical operators last.

**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

```matlab
A == []
```

return an error if A is not 0-by-0 or 1-by-1.

To test for empty arrays, use the function

```matlab
isempty(A)
```
Logical Operators
MATLAB’s logical operators implement the operations AND, OR, and NOT:

\[
\begin{align*}
\& & \text{AND} \\
\vert & \text{OR} \\
\sim & \text{NOT}
\end{align*}
\]

**NOTE** In addition to these logical operators, the `bitfun` directory contains a number of functions that perform bitwise logical operations. See online help for more on these.

MATLAB’s logical operators compare corresponding elements of arrays with equal dimensions. 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 the value 1. Locations where the relation is false receive the value 0.

Each logical operator has a specific set of rules that determines the result of a logical expression:

- An expression using the AND operator, &, is true if both operands are logically true. In numeric terms, the expression is true if both operands are nonzero. For example:
  
  \[
  u = \begin{bmatrix} 1 & 0 & 2 & 3 & 0 & 5 \end{bmatrix}; \\
  v = \begin{bmatrix} 5 & 6 & 1 & 0 & 0 & 7 \end{bmatrix}; \\
  u \& v
  \]

  \[
  \text{ans} = \\
  \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}
  \]

- An expression using the OR operator, |, is true if one operand is logically true, or if both operands are logically true. An OR expression is false only if both
operands are false. In numeric terms, the expression is false only if both operands are zero. Using the u and v above,
\[ u \lor v \]
\[ \text{ans} = \]
\[ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \]

- An expression using the NOT operator, \( \neg \), negates the operand. This produces a false result if the operand is true, and true if it is false. In numeric terms, any nonzero operand becomes zero, and any zero operand becomes one.
\[ \neg u \]
\[ \text{ans} = \]
\[ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \]

**Logical Functions**

In addition to the logical operators, MATLAB provides a number of logical functions. These include:

- The `xor` function performs an exclusive OR on its operands. An exclusive OR expression is `TRUE` if one operand is `TRUE` and the other `FALSE`. In numeric terms, the function returns a 1 if one operand is nonzero and the other is zero. For example,

  \[ a = 1; \]
  \[ b = 1; \]
  \[ \text{xor}(a, b) \]

  \[ \text{ans} = \]
  \[ 0 \]
• The `all` function returns 1 if all the elements of a vector are true, or nonzero. For example,

```matlab
u = [1 2 3 4 0];
if all(u < 3)
    disp('All elements less than three')
end
```

Nothing happens, because `u < 3` is not true for all elements of `u`. Try the same example for

```matlab
u = [0 1 2 0]
```

`all` operates columnwise on matrices. Try entering

```matlab
A = [0 1 2; 3 5 0]
all(A)
```

• The `any` function returns 1 if any of the elements of its argument are nonzero; otherwise, it returns 0. Like `all`, the `any` function operates columnwise on matrices.

• The `isnan` and `isinf` functions return 1s for NaNs and Infs, respectively. The `isfinite` function is true for quantities that are neither inf nor NaN. Try entering

```matlab
A = [0 1 5; 2 NaN -inf];
B = [0 0 15; 2 5 inf];
C = A./B
isfinite(C)
isnan(C)
isinf(C)
```

A number of other MATLAB functions perform logical operations. See the `ops` directory for a complete listing of logical functions.

**Logical Expressions and Subscripting with the find Function**

The `find` function determines the indices of array elements that meet a given logical condition. It’s useful for creating masks and index matrices.
In its most general form, `find` returns a single vector of indices. This vector can index into any size or shape array. For example,

```matlab
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
```

You can also use `find` to obtain both the row and column indices for a rectangular matrix, as well as the array values that meet the logical condition. Use the help facility for more information on `find`.

### Combining Operators in Expressions

You can build expressions that use any combination of arithmetic, logical, and relational operators.

For example, an expression that compares two separate product expressions using the “less than” operator is

```matlab
(a*b) < (c*d)
```

Override the default precedence using parentheses:

```matlab
(a & b) < (c | d)
```
Flow Control

There are four flow control statements in MATLAB:

- `if`, together with `else` and `elseif`, executes a group of statements based on some logical condition.
- `switch`, together with `case` and `otherwise`, executes different groups of statements depending on the value of some logical condition.
- `while` executes a group of statements an indefinite number of times, based on some logical condition.
- `for` executes a group of statements a fixed number of times.

All flow constructs use `end` to indicate the end of the flow control block.

**NOTE**
You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See “Optimizing the Performance of MATLAB Code” on page 10-59 in this chapter.

**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:

```matlab
if logical_expression
    statements
end
```

If the logical expression is 1 (TRUE), MATLAB executes all the statements between the `if` and `end` lines. If the condition is 0 (FALSE), MATLAB skips all the statements between the `if` and `end` lines, and resumes execution at the line following the end statement.

For example,

```matlab
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

```matlab
if X
    statements
end
```

is equivalent to

```matlab
if all(X(:))
    statements
end
```

The else and elseif statements further conditionalize if:

- The else statement has no logical condition. The statements associated with it execute if the preceding if (and possibly elseif condition) is 0 (FALSE).
- The elseif statement has a logical condition that it evaluates if the preceding if (and possibly elseif condition) is 0 (FALSE). The statements associated with it execute if its logical condition is 1 (TRUE). You can have multiple elseifs within an if block.

```matlab
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,

```matlab
if A
    S1
else
    S0
end
```
will execute statement S0 when A is an empty array.

**switch**

*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
    case value2
        statements
    .
    .
    .
    otherwise
        statements
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. `switch` execution ends when MATLAB encounters the next `case` statement or the `otherwise` statement. Only the first matching `case` is executed.
- An `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 case groups. Execution 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 \( \text{value} = \text{expression} \). 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 on screen 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, MATLAB's `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
```
**while**

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is 1 (TRUE). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be 1 (TRUE), 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.

**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:

```matlab
for i = 2:6
    x(i) = 2*x(i-1);
end
```
You can nest multiple `for` loops:

```matlab
for i = 1:m
    for j = 1:n
        A(i,j) = 1/(i + j - 1);
    end
end
```

**NOTE** You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See page 10-59 for details.

**Using Matrices as Indices**

The index of a `for` loop can be an array. For example, consider an `m` by `n` array `A`. The statement

```matlab
for i = A
    statements
end
```
sets `i` equal to the vector `A(:,k)`. For the first loop iteration, `k` is equal to 1; for the second `k` is equal to 2, and so on until `k` equals `n`. That is, the loop iterates for a number of times equal to the number of columns in `A`. For each iteration, `i` is a vector containing one of the columns of `A`. 
Subfunctions

Function M-files can contain code for more than one function. The first function in the file is the primary function, the function invoked with the M-file name. Additional functions within the file are subfunctions that are only visible to the primary function or 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)
    % NEWSTATS Find mean and median with internal functions.
    n = length(u);
    avg = mean(u, n);
    med = median(u, n);
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`. Functions within the same M-file cannot access the same variables unless you declare them as global within the pertinent functions, or pass them as arguments. In addition, the help facility can only access the primary function in an M-file.

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 on your search path. Because it checks for a subfunction first, you can
supersede existing M-files using subfunctions with the same name, for
example, mean in the above code. Function names must be unique within an
M-file, however.

Private Functions

Private functions are functions that reside in subdirectories with the special
name private. They are visible only to functions in the parent directory. 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 of
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.

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.
Indexing and Subscripting

Subscripts
The element in row \( i \) and column \( j \) of \( A \) is denoted by \( A(i, j) \). For example, suppose \( A = \text{magic}(4) \), then \( A(4, 2) \) is the number in the fourth row and second column. For our magic square, \( A(4, 2) \) is 15. So it is possible to compute the sum of the elements in the fourth column of \( A \) by typing

\[
A(1, 4) + A(2, 4) + A(3, 4) + A(4, 4)
\]

It is also possible to refer to the elements of a matrix with a single subscript, \( A(k) \). This is the usual way of referencing row and column vectors. But it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix. So, for our magic square, \( A(8) \) is another way of referring to the value 15 stored in \( A(4, 2) \).

If you try to use the value of an element outside of the matrix, it is an error:

\[
t = A(4, 5)
\]

Index exceeds matrix dimensions

On the other hand, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

\[
x = A;
x(4, 5) = 17
\]

\[
X =
\begin{array}{cccc}
16 & 3 & 2 & 13 & 0 \\
5 & 10 & 11 & 8 & 0 \\
9 & 6 & 7 & 12 & 0 \\
4 & 15 & 14 & 1 & 17 \\
\end{array}
\]

Subscript expressions involving colons refer to portions of a matrix.

\[
A(1:k, j)
\]

is the first \( k \) elements of the \( j \)-th column of \( A \). So

\[
\text{sum}(A(1:4, 4))
\]
computes the sum of the fourth column. But there is a better way. The colon by itself refers to all the elements in a row or column of a matrix and the keyword end refers to the last row or column. So

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

computes the sum of the elements in the last column of A.

```
ans =
   34
```

**Concatenation**

Concatenation is the process of joining small matrices together to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [], is the concatenation operator. For an example, start with the 4-by-4 magic square, A, and form

```
B = [ A A+32; A+48 A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =
   16 3  2 13 48 35 34 45
   5 10 11  8 37 42 43 40
   9  6  7 12 41 38 39 44
   4 15 14  1 36 47 46 33
  64 51 50 61 32 19 18 29
  53 58 59 56 21 26 23 28
  57 54 55 60 25 22 23 28
  52 63 62 49 20 31 30 17
```

This matrix is half way to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square.

```
sum(B)
```

```
ans =
   260 260 260 260 260 260 260 260
```

But its row sums, `sum(B')`, are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.
Deleting Rows and Columns
You can delete rows and columns from a matrix using just a pair of square brackets. Start with
\[ X = A; \]
Then, to delete the second column of \( X \) use
\[ X(:,2) = []; \]
This changes \( X \) to
\[
\begin{bmatrix}
16 & 2 & 13 \\
5 & 11 & 8 \\
9 & 7 & 12 \\
4 & 14 & 1 \\
\end{bmatrix}
\]
If you delete a single element from a matrix, the result isn’t a matrix anymore. So, expressions like
\[ X(1,2) = []; \]
result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So
\[ X(2:2:10) = []; \]
results in
\[
\begin{bmatrix}
16 & 9 & 2 & 7 & 13 & 12 & 1 \\
\end{bmatrix}
\]
Advanced Indexing
MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended end to end. For example, MATLAB stores
\[
A = [2 6 9; 4 2 8; 3 0 1]
\]
Accessing A with a single subscript indexes directly into the storage column. \( A(3) \) accesses the third value in the column, the number 3. \( A(7) \) accesses the seventh value, 9, and so on.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like \( A \) has size \([d_1\ d_2]\), where \( d_1 \) is the number of rows in the array and \( d_2 \) is the number of columns. If you supply two subscripts \((i, j)\) representing row-column indices, the offset is

\[(j - 1) \times d_1 + i\]

Given the expression \( A(3, 2) \), MATLAB calculates the offset into \( A \)'s storage column as \((2-1)\times3+3\), or 6. Counting down 6 elements in the column accesses the value 0.

This storage and indexing scheme 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. For example, consider a 5-by-4-by-3-by-2 array $C$:

MATLAB displays $C$ as

\[
\begin{array}{cccc}
\text{page}(1,1) = & 1 & 4 & 3 & 5 \\
& 1 & 7 & 9 \\
& 6 & 3 & 2 \\
& 1 & 5 & 9 \\
& 2 & 7 & 5 \\
\end{array}
\]

MATLAB stores \( (1 2) \):

\[
\begin{array}{cccc}
1 & 2 & 5 & 0 \\
4 & 1 & 6 & 1 \\
3 & 2 & 7 & 5 \\
5 & 9 & 2 & 6 \\
9 & 5 & 3 & 1 \\
\end{array}
\]
Again, a single subscript indexes directly into this column. For example, C(4) produces the result

\[
\text{ans} = 0
\]

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.

Valid Subscripts for 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 \([d_1 \ d_2 \ d_3 \ d_4]\). MATLAB calculates the offset into the storage column by

\[
(i-1)(d_3)(d_2)(d_1)+(k-1)(d_2)(d_1)+(j-1)(d_1)+i
\]

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

\[
(s_{n-1})(d_{n-1})(d_{n-2})\ldots(d_1)+(s_{n-2})(d_{n-3})\ldots(d_1)+\ldots+(s_2-1)(d_1)+s_1
\]

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

\[
C(3, 2, 1, 1, 1, 1, 1)
\]

is equivalent to

\[
C(3, 2)
\]
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 “on the fly.”

**eval**
The `eval` function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the `eval` syntax is

```matlab
eval('string')
```

For example, this code uses `eval` on an expression to generate a Hilbert matrix of order `n`:

```matlab
t = '1/(i+j-1)';
for i = 1:n
    for j = 1:n
        a(i,j) = eval(t);
    end
end
```

Here's an example that uses `eval` on a statement:

```matlab
eval('t = clock')
```

**feval**
`feval` differs from `eval` in that it executes a function whose name is in a string, rather than an entire MATLAB expression. You can use `feval` and the `input` function to choose one of several tasks defined by M-files. In this example, the functions have names like `sin`, `cos`, and so on.

```matlab
fun = ['sin'; 'cos'; 'log'];
k = input('Choose function number: ');
x = input('Enter value: ');
feval(fun(k,:),x)
```
**NOTE** Use **feval** rather than **eval** whenever possible. M-files that use **feval** execute faster and can be compiled with the MATLAB compiler.

---

**Constructing Strings for Evaluation**

You can concatenate strings to create a complete expression for input to **eval**. This code shows how **eval** can create ten variables named P1, P2, ...P10, and set each of them to a different value:

```matlab
for i=1:10
    eval(['P',int2str(i),'= i.^2'])
end
```
Command/ Function Duality

MATLAB commands are statements like

```matlab
load
help p
```

Many commands accept modifiers that specify operands.

```matlab
load August17.dat
help magic
type rank
```

An alternate method of supplying the command modifiers makes them string arguments of functions.

```matlab
load('August17.dat')
help('magic')
type('rank')
```

This is MATLAB's "command/function duality." Any command of the form

```matlab
command argument
```

can also be written in the functional form

```matlab
command('argument')
```

The advantage of the functional approach comes when the string argument is constructed from other pieces. The following example processes multiple data files, August1.dat, August2.dat, and so on. It uses the function int2str, which converts an integer to a character string, to help build the filename.

```matlab
for d = 1:31
    s = ['August' int2str(d) '.dat']
    load(s)
    % Process the contents of the d-th file
end
```
Empty Matrices

Earlier versions of MATLAB allowed for only one empty matrix, the 0-by-0 array denoted by []. MATLAB 5 provides for matrices and arrays where one, but not all, of the dimensions is zero. For example, 1-by-0, 10-by-0-by-20, and [3 4 0 5 2] are all possible array sizes:

The two-character sequence [ ] continues to denote the 0-by-0 matrix. Empty arrays of other sizes can be created with the functions zeros, ones, rand, or eye. To create a 0-by-5 matrix, for example, use,

\[ E = \text{zeros}(0,5) \]

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 should be that same function, 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.

Many operations in MATLAB produce row vectors or column vectors. It is possible for the result to be the empty row vector,

\[ r = \text{zeros}(1,0) \]

or the empty column vector

\[ C = \text{zeros}(0,1) \]

MATLAB 5 retains MATLAB 4 behavior for if and while statements. For example

\[ \text{if } A, S1, \text{ else, } S0, \text{ end } \]

executes statement \( S0 \) when \( A \) is an empty array.

Some MATLAB functions, like sum and max, are reductions. For matrix arguments, these functions produce vector results; for vector arguments they
Empty matrices produce scalar results. Empty inputs produce the following results with these functions:

- \texttt{sum(\[]\)} is 0
- \texttt{prod(\[]\)} is 1
- \texttt{max(\[]\)} is \[\]
- \texttt{min(\[]\)} is \[\]
Errors and Warnings

In many cases, it's desirable to take specific actions when different kinds of errors occur. For example, you may want to prompt the user for more input, display extended error or warning information, or repeat a calculation using default values. MATLAB’s error handling capabilities let your application check for particular error conditions and execute appropriate code depending on the situation.

Error Handling with eval and lasterr

The basic tools for error-handling in MATLAB are

- The `eval` function, which lets you execute a function and specify a second function to execute if an error occurs in the first.
- The `lasterr` function, which returns a string containing the last error generated by MATLAB.

The `eval` function provides error-handling capabilities using the two argument form

```matlab
eval ('trystring','catchstring')
```

If the operation specified by `trystring` executes properly, `eval` simply returns. If `trystring` generates an error, the function evaluates `catchstring`. Use `catchstring` to specify a function that determines the error generated by `trystring` and takes appropriate action.

The `try/catch` `eval` form is especially useful in conjunction with the `lasterr` function. `lasterr` returns a string containing the last error message generated by MATLAB. Use `lasterr` inside the `catchstring` function to “catch” the error generated by `trystring`.

For example, this function uses `lasterr` to check for a specific error message that can occur during matrix multiplication. The error message indicates that matrix multiplication is impossible because the operands have different inner...
dimensions. If the message occurs, the code truncates one of the matrices to perform the multiplication:

```matlab
function C = catch(A, B)
  l = lasterr;
  j = findstr(l, 'Inner matrix dimensions')
  if (~isempty(j))
    [m n] = size(A)
    [p q] = size(B)
    if (n>p)
      A(:, p+1:n) = []
    elseif (n<p)
      B(n+1:p, :) = []
    end
    C = A*B;
  else
    C = 0;
  end

Using the two-argument form of eval with the catch function shown above:

```matlab
clear
A = [1 2 3; 6 7 2; 0 1 5];
B = [9 5 6; 0 4 9];
eval('A*B', 'catch(A, B)')

A = 1:7;
B = randn(9, 9);
eval('A*B', 'catch(A, B)')
```

**Displaying Error and Warning Messages**

Use the `error` and `fprintf` functions to display error information on the screen. The `error` function has the syntax

```matlab
error('error string')
```
If you call the `error` function from inside an M-file, `error` displays the text in the quoted string and causes the M-file to stop executing. For example, suppose the following appears inside the M-file `myfile.m`:

```matlab
if n < 1
    error('n must be 1 or greater. ')
end
```

For `n` equal to 0, the following text appears on the screen and the M-file stops:

```matlab
??? Error using => myfile
 n must be 1 or greater.
```

In MATLAB, warnings are similar to error messages, except program execution does not stop. Use the `warning` function to display warning messages,

```matlab
warning('warning string')
```
MATLAB provides functions for time and date handling. These functions are in a directory called `timefun` in the MATLAB toolbox.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current time and date</td>
<td><code>now</code></td>
<td>Current date and time as serial date number.</td>
</tr>
<tr>
<td></td>
<td><code>date</code></td>
<td>Current date as date string.</td>
</tr>
<tr>
<td></td>
<td><code>clock</code></td>
<td>Current date and time as date vector.</td>
</tr>
<tr>
<td>Conversion</td>
<td><code>datenum</code></td>
<td>Convert to serial date number.</td>
</tr>
<tr>
<td></td>
<td><code>datestr</code></td>
<td>Convert to string representation of date.</td>
</tr>
<tr>
<td></td>
<td><code>datevec</code></td>
<td>Date components.</td>
</tr>
<tr>
<td>Utility</td>
<td><code>calendar</code></td>
<td>Calendar.</td>
</tr>
<tr>
<td></td>
<td><code>weekday</code></td>
<td>Day of the week.</td>
</tr>
<tr>
<td></td>
<td><code>eomday</code></td>
<td>End of month.</td>
</tr>
<tr>
<td></td>
<td><code>datetick</code></td>
<td>Date formatted tick labels.</td>
</tr>
<tr>
<td>Timing</td>
<td><code>cputime</code></td>
<td>CPU time in seconds.</td>
</tr>
<tr>
<td></td>
<td><code>tic</code>, <code>toc</code></td>
<td>Stopwatch timer.</td>
</tr>
<tr>
<td></td>
<td><code>etime</code></td>
<td>Elapsed time.</td>
</tr>
</tbody>
</table>

**Date Formats**

MATLAB works with three different date formats: date strings, serial date numbers, and date vectors.

When dealing with dates you typically work with date strings (16-Sep-1996). MATLAB works internally with serial date numbers (729284). A serial date 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 '16-Sep-1996, 6:00 pm' in MATLAB is date number 729284.75.

All functions that require dates accept either date strings or serial date numbers. If you are dealing with a few dates at the MATLAB command-line level, date strings are more convenient. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you will get better performance if you use date numbers.

Date vectors are an internal format for some MATLAB functions and you will not typically use them in calculations. A date vector contains the elements [year month day hour minute second].

MATLAB provides functions that convert date strings to serial date numbers, and vice versa. Dates can also be converted to date vectors.

Here are examples of the three date formats used by MATLAB:

<table>
<thead>
<tr>
<th>Serial date number</th>
<th>729300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date string</td>
<td>02-Oct-1996</td>
</tr>
<tr>
<td>Date vector</td>
<td>1996 10 2 0 0 0</td>
</tr>
</tbody>
</table>

**Conversions Between Date Formats**

Functions that convert between date formats are:

- **datenum**: Convert date string to serial date number
- **datestr**: Convert serial date number to date string
- **datevec**: Split date number or date string into individual date elements
Here are some examples of conversions from one date format to another:

```matlab
d1 = datenum('02-Oct-1996')
d1 =
    729300

d2 = datenum(d1+10)
d2 =
    12-Oct-1996
dv1 = datevec(d1)
dv1 =
    199610 2 0 0 0

dv2 = datevec(d2)
dv2 =
    199610 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 non-negative 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 function `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.

<table>
<thead>
<tr>
<th>dateform</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01- Mar - 1996 15:45:17</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>1</td>
<td>01- Mar - 1996</td>
<td>day-month-year</td>
</tr>
<tr>
<td>2</td>
<td>03/01/96</td>
<td>month/day/year</td>
</tr>
<tr>
<td>3</td>
<td>Mar</td>
<td>month, three letters</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>month, single letter</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>month</td>
</tr>
<tr>
<td>6</td>
<td>03/01</td>
<td>month/day</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>day of month</td>
</tr>
<tr>
<td>8</td>
<td>Wed</td>
<td>day of week, three letters</td>
</tr>
<tr>
<td>9</td>
<td>W</td>
<td>day of week, single letter</td>
</tr>
<tr>
<td>10</td>
<td>1996</td>
<td>year, four digits</td>
</tr>
<tr>
<td>11</td>
<td>96</td>
<td>year, two digits</td>
</tr>
<tr>
<td>12</td>
<td>Mar 96</td>
<td>month year</td>
</tr>
<tr>
<td>13</td>
<td>15:45:17</td>
<td>hour:minute:second</td>
</tr>
<tr>
<td>14</td>
<td>03:45:17 PM</td>
<td>hour:minute:second AM or PM</td>
</tr>
<tr>
<td>15</td>
<td>15:45</td>
<td>hour:minute</td>
</tr>
<tr>
<td>16</td>
<td>03:45 PM</td>
<td>hour:minute AM or PM</td>
</tr>
<tr>
<td>17</td>
<td>Q1-96</td>
<td>calendar quarter-year</td>
</tr>
<tr>
<td>18</td>
<td>Q1</td>
<td>calendar quarter</td>
</tr>
</tbody>
</table>

d = '01-Mar-1996'

d =

01-Mar-1996

datestr(d)

ans =

01-Mar-1996

datestr(d, 2)

ans =

03/01/96
Current Date and Time
The function `date` returns a string for today's date.

```matlab
date
ans =
    Q1-96
```

The function `now` returns the serial date number for the current date and time.

```matlab
now
ans =
    729300.71
```

```matlab
datestr(now)
ans =
    02-Oct-1996 16:56:16
```

```matlab
datestr(floor(now))
ans =
    02-Oct-1996
```
Obtaining User Input

To obtain input from a user during M-file execution, you can

- Display a prompt and obtain keyboard input.
- Pause until the user presses a key.
- Build a complete graphical user interface.

This section covers the first two topics. The third topic is discussed in Using MATLAB Graphics and Building GULs with MATLAB.

Prompting for Keyboard Input

The `input` function displays a prompt and waits for a user response. Its syntax is

```
 n = input('prompt_string')
```

The function displays the `prompt_string`, waits, and then returns the value input from the keyboard. If the user inputs an expression, the function evaluates it and returns its value. This function is useful for implementing menu-driven applications.

`input` can also return user input as a string, rather than a numeric value. To obtain string input, append `'s'` to the function's argument list:

```
 name = input('Enter address: ','s');
```

Pausing During Execution

Some kinds of M-files benefit from pauses between execution steps. For example, the `petals.m` script shown on page 10-5 pauses between the plots it creates, allowing the user to display a plot for as long as desired and then press a key to move to the next plot.

The `pause` command, with no arguments, stops execution until the user presses a key. To pause for `n` seconds, use:

```
 pause(n)
```
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:

```matlab
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 to obtain the results.

Reading and Writing MAT-Files

The MAT-file subroutine library, described in the MATLAB Application Program Interface Guide, provides routines for reading and writing MAT-files. Also see the MATLAB Application Program Interface Guide for information on MEX-files, C or Fortran functions that you can call directly from MATLAB.
Optimizing the Performance of MATLAB Code

This section describes techniques that often improve the execution speed and memory management of MATLAB code:

- Vectorization of loops
- Vector preallocation

MATLAB is a matrix language, which means it is designed for vector and matrix operations. For best performance, you should take advantage of this where possible.

Vectorization of Loops

You can speed up your M-file code by vectorizing algorithms. Vectorization means converting for and while loops to equivalent vector or matrix operations.

A Simple Example

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```matlab
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

A vectorized version of the same code is

```matlab
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, then using the tic and toc commands to time the M-files.

An Advanced Example

`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:

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix};
\]

\[
B = \text{repmat}(A, 2, 3);
\]

\[
B =
\begin{bmatrix}
1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 & 5 & 6 \\
1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 & 5 & 6
\end{bmatrix}
\]

`repmat` uses vectorization to create the indices that place elements in the output array.

```matlab
function B = repmat(A, M, N)
if nargin < 2
    error('Requires at least 2 inputs.'
elseif nargin == 2
    N = M;
end
[m, n] = size(A);
mi nd = (1: m)';
ni nd = (1: n)';
mi nd = mi nd(:, ones(1, M));
ni nd = ni nd(:, ones(1, N));
B = A(mi nd, ni nd);
```

Section 1 above obtains the row and column sizes of the input array.

Section 2 creates two column vectors. `mi nd` contains the integers from 1 through the row size of `A`. The `ni nd` variable contains the integers from 1 through the column size of `A`.

Section 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

\[
B = A(:, \text{ones}(1, n\_cols))
\]

where `n\_cols` is the desired number of columns in the resulting matrix.
The final line of \texttt{repmat} uses array indexing to create the output array. Each element of the row index array, \texttt{mind}, is paired with each element of the column index array, \texttt{nind}:

1. The first element of \texttt{mind}, the row index, is paired with each element of \texttt{nind}. MATLAB moves through the \texttt{nind} matrix in a columnwise fashion, so \texttt{mind(1,1)} goes with \texttt{nind(1,1)}, then \texttt{nind(2,1)}, and so on. The result fills the first row of the output array.

2. Moving columnwise through \texttt{mind}, each element is paired with the elements of \texttt{nind} as above. Each complete pass through the \texttt{nind} matrix fills one row of the output array.

\section*{Array Preallocation}

You can often improve code execution time by preallocating the arrays that store output results. Preallocation prevents MATLAB from having to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Array Type} & \textbf{Function} & \textbf{Examples} \\
\hline
Numeric array & zeros & \begin{verbatim}
y = zeros(1,100)
for i = 1:100
    y(i) = det(X^i);
end
\end{verbatim} \\
\hline
Cell array & cell & \begin{verbatim}
B = cell(2,3)
B{1,3} = 1:3;
B{2,2} = 'string';
\end{verbatim} \\
\hline
Structure array & struct & \begin{verbatim}
data = struct([1 3], x', [1 3], y', [5 6])
data(3).x = [9 0 2];
data(3).y = [5 6 7];
\end{verbatim} \\
\hline
\end{tabular}
\end{table}

Preallocation also helps reduce memory fragmentation if you work with large matrices. In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. This can result in plenty of free memory, but not enough contiguous space to hold a large variable.
Preallocation helps prevent this by allowing MATLAB to “grab” sufficient space for large data constructs at the beginning of a computation.

**Notes on Memory Use**

This section discusses ways to conserve memory and improve memory use.

**Memory Management Functions**

MATLAB has five functions to improve how memory is handled:

- **clear** removes variables from memory.
- **pack** saves existing variables to disk, then reloads them contiguously. Because of time considerations, you should not use **pack** within loops or M-file functions.
- **quit** exits MATLAB and returns all allocated memory to the system.
- **save** selectively stores variables to disk.
- **load** reloads a data file saved with the **save** command.

**NOTE**  
**save** and **load** are faster than MATLAB low-level file I/O routines. **save** and **load** have been optimized to run faster and reduce memory fragmentation. See Chapter 2, “Using the Environment” for details on these functions.

On some systems, the **who** command displays the amount of free memory remaining. However, be aware that

- If you delete a variable from the workspace, the amount of free memory indicated by **who** usually does not get larger unless the deleted variable occupied the highest memory addresses. The number actually indicates the amount of contiguous, unused memory. Clearing the highest variable makes the number larger, but clearing a variable beneath the highest variable has no effect. This means that you might have more free memory than is indicated by **who**.
- Computers with virtual memory do not display the amount of free memory remaining because neither MATLAB nor the hardware imposes limitations.
Removing a Function From Memory
MATLAB creates a list of M- and MEX-filenames at startup for all files that reside below the `matlab/toolbox` directories. This list is stored in memory and is freed only when a new list is created during a call to the `path` function. Function M-file code and MEX-file relocatable code are loaded into memory when the corresponding function is called. The M-file code or relocatable code is removed from memory when:

- The function is called again and a new version now exists.
- The function is explicitly cleared with the `clear` command.
- All functions are explicitly cleared with the `clear functions` command.
- MATLAB runs out of memory.

Nested Function Calls
The amount of memory used by nested functions is the same as the amount used by calling them on consecutive lines. These two examples require the same amount of memory:

```matlab
result = function2(function1(input99));
result = function1(input99);
result = function2(result);
```

Variables and Memory
Memory is allocated for variables whenever the left-hand side variable in an assignment does not exist. The statement

```matlab
x = 10
```

allocates memory, but the statement

```matlab
x(10) = 1
```

does not allocate memory if the 10th element of `x` exists.
To conserve memory:

is not preferred because $y$ is both an input and an output variable.

- Avoid creating large temporary variables, and clear temporary variables when they are no longer needed.
- Set variables equal to the empty matrix $[]$ to free memory, or clear them using
  
  ```matlab
  clear variable_name
  ```

- Reuse variables as much as possible.

**Global Variables.** Declaring variables as `global` merely puts a flag in a symbol table. It does not use any more memory than defining nonglobal variables. Consider the following example:

```matlab
global a
a = 5;
```

Now there is one copy of $a$ stored in the MATLAB workspace. Typing

```matlab
clear a
```

removes $a$ from the MATLAB workspace, but it still exists in the global workspace.

```matlab
clear global a
```

removes $a$ from the global workspace.

**PC-Specific Topics**

- There are no functions implemented to manipulate the way MATLAB handles 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 Uicontrols. The best way to free up system resources is to close all inactive windows. Iconified windows still use resources.
- The performance of a permanent swap file is typically better than a temporary swap file.
- Typically a swap file twice the size of the installed RAM is sufficient.
UNIX-Specific Topics

- Memory that MATLAB requests from the operating system is not returned to the operating system until the MATLAB process is finished.
- MATLAB 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 in the heap 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:

```matlab
a = rand(1e6, 1);
b = rand(1e6, 1);
```

These use approximately 16.4 MB of RAM:

```matlab
c = rand(2.1e6, 1);
```

And these use approximately 32.4 MB of RAM:

```matlab
a = rand(1e6, 1);
b = rand(1e6, 1);
clear
c = rand(2.1e6, 1);
```

This is because MATLAB is not able to fit a 2.1 MB array in the space previously occupied by two 1 MB arrays. The simplest way to prevent overallocation of memory, is to preallocate the largest vector. This series of statements uses approximately 32.4 MB of RAM:

```matlab
a = rand(1e6, 1);
b = rand(1e6, 1);
clear
c = rand(2.1e6, 1);
```

while these use only about 16.4 MB of RAM:

```matlab
c = rand(2.1e6, 1);
clear
a = rand(1e6, 1);
b = rand(1e6, 1);
```

Allocating the largest vectors first allows for optimal use of the available memory.
What Does “Out of Memory” Mean?

Typically the Out of Memory message appears because MATLAB asked the operating system for a segment of memory larger than that which is currently available. Use any of the techniques discussed in this section to help optimize the available memory. If the Out of Memory message still appears consistently:

- Increase the size of the swap file.
- Make sure that there are no external constraints on the memory accessible to MATLAB (on UNIX systems use the limit command to check).
- Add more memory to the system.
- Reduce the size of the data you are using.
Character Arrays (Strings)

Character Arrays ....... 11-4
Converting Between ASCII Characters and Values .... 11-5
Creating Two-Dimensional Character Arrays .... 11-5

Cell Arrays of Strings ....... 11-7
Converting Between Character Arrays and Cell Arrays of Strings .... 11-7

String Comparisons ....... 11-9
Comparing Strings For Equality .... 11-9
Comparing Characters for Equality with Operators .... 11-10
Categorizing Characters Within a String .... 11-11

Searching and Replacing .... 11-12

String/Numeric Conversion .... 11-13
Array/String Conversion .... 11-14
This chapter explains MATLAB’s support for string data. It describes how to create character arrays and cell arrays of strings, the two ways to represent strings. It also discusses how to perform common string operations, such as searching and replacing, and how to convert between string and numeric formats.

The string functions are located in the directory named `strfun` in the MATLAB toolbox.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>char</td>
<td>Create character array (string).</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>Convert string to numeric codes.</td>
</tr>
<tr>
<td></td>
<td>cellstr</td>
<td>Create cell array of strings from character array.</td>
</tr>
<tr>
<td></td>
<td>blanks</td>
<td>String of blanks.</td>
</tr>
<tr>
<td></td>
<td>deblank</td>
<td>Remove trailing blanks.</td>
</tr>
<tr>
<td></td>
<td>eval</td>
<td>Execute string with MATLAB expression.</td>
</tr>
<tr>
<td>String Tests</td>
<td>ischar</td>
<td>True for character array.</td>
</tr>
<tr>
<td></td>
<td>iscellstr</td>
<td>True for cell array of strings.</td>
</tr>
<tr>
<td></td>
<td>isletter</td>
<td>True for letters of alphabet.</td>
</tr>
<tr>
<td></td>
<td>isspace</td>
<td>True for whitespace characters.</td>
</tr>
<tr>
<td>String Operations</td>
<td>strcat</td>
<td>Concatenate strings.</td>
</tr>
<tr>
<td></td>
<td>strvcat</td>
<td>Concatenate strings vertically.</td>
</tr>
<tr>
<td></td>
<td>strcmp</td>
<td>Compare strings.</td>
</tr>
<tr>
<td></td>
<td>strncmp</td>
<td>Compare first N characters of strings.</td>
</tr>
<tr>
<td></td>
<td>findstr</td>
<td>Find one string within another.</td>
</tr>
<tr>
<td></td>
<td>strjust</td>
<td>Justify string.</td>
</tr>
<tr>
<td></td>
<td>strmatch</td>
<td>Find matches for string.</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>strrep</td>
<td>Replace string with another.</td>
</tr>
<tr>
<td></td>
<td>strtok</td>
<td>Find token in string.</td>
</tr>
<tr>
<td></td>
<td>upper</td>
<td>Convert string to uppercase.</td>
</tr>
<tr>
<td></td>
<td>lower</td>
<td>Convert string to lowercase.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String to Number</td>
<td>num2str</td>
<td>Convert number to string.</td>
</tr>
<tr>
<td>Conversion</td>
<td>int2str</td>
<td>Convert integer to string.</td>
</tr>
<tr>
<td></td>
<td>mat2str</td>
<td>Convert matrix to eval 'able string.</td>
</tr>
<tr>
<td></td>
<td>str2num</td>
<td>Convert string to number.</td>
</tr>
<tr>
<td></td>
<td>sprintf</td>
<td>Write formatted data to string.</td>
</tr>
<tr>
<td></td>
<td>sscanf</td>
<td>Read string under format control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Number</td>
<td>hex2num</td>
<td>Convert IEEE hexadecimal to double precision number.</td>
</tr>
<tr>
<td>Conversion</td>
<td>hex2dec</td>
<td>Convert hexadecimal string to decimal integer.</td>
</tr>
<tr>
<td></td>
<td>dec2hex</td>
<td>Convert decimal integer to hexadecimal string.</td>
</tr>
<tr>
<td></td>
<td>bin2dec</td>
<td>Convert binary string to decimal integer.</td>
</tr>
<tr>
<td></td>
<td>dec2bin</td>
<td>Convert decimal integer to binary string.</td>
</tr>
<tr>
<td></td>
<td>base2dec</td>
<td>Convert base B string to decimal integer.</td>
</tr>
<tr>
<td></td>
<td>dec2base</td>
<td>Convert decimal integer to base B string.</td>
</tr>
</tbody>
</table>
Character Arrays

In MATLAB, the term string refers to an array of characters. MATLAB represents each character internally as its corresponding ASCII value. Unless you want to access these values, however, you can simply work with the characters as they display on screen.

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:

```matlab
name = 'Thomas R. Lee';
```

In the workspace, the output of `whos` shows

```
Name       Size         Bytes  Class
name       1x13            26  char array
```

You can see that a character uses two bytes of storage internally.

The `class` and `ischar` functions show `name`'s identity as a character array,

```matlab
class(name)
```

```
ans =
char
```

```matlab
ischar(name)
```

```
an =
1
```
Converting Between ASCII Characters and Values
Character arrays store each character as a 16-bit ASCII value. Use the `double` function to convert strings to their ASCII numeric values, and `char` to revert to character representation:

```matlab
name = double(name)
```

```
name =
Columns 1 through 12
     84   104   111   109    97   115    32    82    46    32    76   101

Column 13
101
```

```matlab
name = char(name)
```

```
Thomas R. Lee
```

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:

```matlab
name = ['Thomas R. Lee' ; 'Sr. Developer']
```

```
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:

```matlab
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 this
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:

```matlab
name = char('Thomas R. Lee','Senior Developer')
```

```matlab
name =

Thomas R. Lee
Senior Developer
```

When extracting strings from an array, use the `deblank` function to remove any trailing blanks:

```matlab
trimname = deblank(name(1,:))
```

```matlab
trimname =

Thomas R. Lee
```

```matlab
size(trimname)
```

```matlab
ans =

1    13
```
Cell Arrays of Strings

It’s often convenient to store groups of strings in cell arrays instead of standard character arrays. This prevents you from having to pad strings with blanks to create character arrays with rows of equal length. A set of functions enables you to work with cell arrays of strings:

• You can convert between standard character arrays and cell arrays of strings.
• You can apply string comparison operations to cell arrays of strings.

For details on cell arrays see the Structures and Cell Arrays chapter.

Converting Between Character Arrays and Cell Arrays of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider the character array

```matlab
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:

```matlab
celldata = cellstr(data)
```

```matlab
celldata =
'Allison Jones'
'Development'
'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix:

```matlab
length(celldata{3})
```

```matlab
ans =
7
```
Use char to convert back to a standard padded character array:

```
strings = char(celldata)

strings =

Allison Jones
Development
Phoenix
```
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 whitespace.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings For Equality

There are two functions that 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.

Consider the two strings:

```matlab
str1 = 'hello';
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns 0 (false), for example:

```matlab
C = strcmp(str1,str2)
```

```matlab
C =
0
```

NOTE FOR C PROGRAMMERS  This is an important difference between MATLAB’s `strcmp` and C’s `strcmp()`, which for unfortunate historical reasons returns 0 if the two strings are the same.
The first three characters of \texttt{str1} and \texttt{str2} are identical, so invoking \texttt{strncmp} with any value up to 3 returns 1:

\begin{verbatim}
C = strncmp(str1,str2,2)
\end{verbatim}

\begin{verbatim}
C = 1
\end{verbatim}

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings:

\begin{verbatim}
A = {'pizza';'chips';'candy'};
B = {'pizza';'chocolate';'pretzels'};
\end{verbatim}

Now apply the string comparison functions:

\begin{verbatim}
strcmp(A,B)
\end{verbatim}

\begin{verbatim}
ans =
1
0
0
\end{verbatim}

\begin{verbatim}
strncmp(A,B,1)
\end{verbatim}

\begin{verbatim}
ans =
1
1
0
\end{verbatim}

### Comparing Characters for Equality with 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 which characters in two strings match:

```
A = 'fate';
B = 'cake';
A == B
```

```
ans =
  0 1 0 1
```

All of the relational operators (>, >=, <, <=, ==, !=) compare the ASCII values of corresponding characters.

**Categorizing Characters Within a String**

There are two functions for categorizing characters inside a string:

- `isletter` determines if a character is a letter
- `isspace` determines if a character is whitespace (blank, tab, or new line)

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 1 (true) because the first four characters of `mystring` are letters.
MATLAB provides several functions for searching and replacing characters in a string. 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`,

```
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 whitespace characters. You can use the `strtok` function to parse a sentence into words; for example:

```
function all_words = words(input_string)
    remainder = input_string;
    all_words = '';

    while (any(remainder))
        [chopped, remainder] = strtok(remainder);
        all_words = strvcat(all_words, chopped);
    end
```

String/ Numeric Conversion

MATLAB's string/numeric conversion functions change numeric values into character strings. You can store numeric values as digit-by-digit string representations, or convert a value into a hexadecimal or binary string. Consider a scalar \( x \):

\[
x = 5317;
\]

By default, MATLAB stores the number \( x \) as a 1-by-1 double array containing the value 5317. The \texttt{int2str} (integer to string) function breaks this scalar into a 1-by-4 vector containing the string '5317':

\[
y = \text{int2str}(x);
size(y)
\]

\[
\text{ans} =
\begin{array}{c}
1 \\
4
\end{array}
\]

A related function, \texttt{num2str}, provides more control over the format of the output string. An optional second argument sets the number of digits in the output string, or specifies an actual format.

\[
p = \text{num2str}(\pi, 9)
\]

\[
p =
3.14159265
\]

Both \texttt{int2str} and \texttt{num2str} are handy for labeling plots. For example, the following lines use \texttt{num2str} to prepare automated labels for the \( x \)-axis of a plot:

\[
\text{function plotLabel}(x, y)
\begin{align*}
\text{plot}(x, y); \\
\text{str1} = \text{num2str}(	ext{min}(x)); \\
\text{str2} = \text{num2str}(	ext{max}(x)); \\
\text{out} &= \left[ '\text{Value of } f \text{ from } ' \str1 \text{ to } ' \str2 \right]; \\
\text{xlabel}(\text{out});
\end{align*}
\]

Another class of numeric/string conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or...
hexadecimal representation. For example, the `dec2hex` function converts a decimal value into the corresponding hexadecimal string:

```
dec_num = 4035
hex_num = dec2hex(dec_num)
```

```
hex_num =

FC3
```

See the `strfun` directory for a complete listing of string conversion functions.

## Array/ String Conversion

The MATLAB function `mat2str` changes an array to a string that MATLAB can evaluate. This string is useful input for a function such as `eval`, which evaluates input strings just as if they were typed at the MATLAB command line.

Create a 3-by-2 array `A`:

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

```
A =

1     2     3
4     5     6
```

`mat2str` returns a string that contains the text you would enter to create `A` at the command line:

```
B = mat2str(A)
```

```
B =

[ 1 2 3; 4 5 6]
```
# Multidimensional Arrays

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Arrays</td>
<td>12-3</td>
</tr>
<tr>
<td>Creating Multidimensional Arrays</td>
<td>12-4</td>
</tr>
<tr>
<td>Getting Information About Multidimensional Arrays</td>
<td>12-9</td>
</tr>
<tr>
<td>Working with Multidimensional Arrays</td>
<td>12-9</td>
</tr>
<tr>
<td>Indexing</td>
<td>12-10</td>
</tr>
<tr>
<td>Reshaping</td>
<td>12-11</td>
</tr>
<tr>
<td>Permuting Array Dimensions</td>
<td>12-13</td>
</tr>
</tbody>
</table>

**Computation with Multidimensional Arrays** 12-15
- Functions that Operate on Vectors 12-15
- Functions that Operate Element-by-Element 12-15
- Functions that Operate on Planes and Matrices 12-16

**Organizing Data in Multidimensional Arrays** 12-17

**Multidimensional Cell Arrays** 12-19

**Multidimensional Structure Arrays** 12-20
- Applying Functions to Multidimensional Structure Arrays 12-21
This chapter discusses multidimensional arrays, MATLAB arrays with more than two dimensions. Multidimensional arrays can be numeric, character, cell, or structure arrays.

Multidimensional arrays are broadly useful—for example, in the representation of multivariate data, or multiple pages of two-dimensional data. MATLAB provides a number of functions that directly support multidimensional arrays. You can extend this support by creating M-files that work with your data architecture.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>Concatenate arrays.</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>ndgrid</td>
<td>Generate arrays for N-D functions and interpolation.</td>
</tr>
<tr>
<td>permute</td>
<td>Permute array dimensions.</td>
</tr>
<tr>
<td>ipermute</td>
<td>Inverse permute array dimensions.</td>
</tr>
<tr>
<td>shiftdim</td>
<td>Shift dimensions.</td>
</tr>
<tr>
<td>squeeze</td>
<td>Remove singleton dimensions.</td>
</tr>
</tbody>
</table>
Multidimensional Arrays

Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.

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 guide 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 \((2, 3, 2)\).

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.

**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.
- Generating arrays using MATLAB functions.
- Using the `cat` function to build multidimensional arrays.
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 \( A \):

\[
A = \begin{bmatrix}
5 & 7 & 8 \\
0 & 1 & 9 \\
4 & 3 & 6
\end{bmatrix}.
\]

\( A \) 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 \( A \),

\[
A(:, :, 2) = \begin{bmatrix}
1 & 0 & 4 \\
3 & 5 & 6 \\
9 & 8 & 7
\end{bmatrix}
\]

MATLAB responds with

\[
A(:, :, 1) = \\
\begin{bmatrix}
5 & 7 & 8 \\
0 & 1 & 9 \\
4 & 3 & 6
\end{bmatrix}
\]

\[
A(:, :, 2) = \\
\begin{bmatrix}
1 & 0 & 4 \\
3 & 5 & 6 \\
9 & 8 & 7
\end{bmatrix}
\]

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 MATLAB’s scalar expansion capabilities, together with the colon operator, to fill an entire dimension with a single value.

\[
A(:,:,3) = 5 \\
A(:,:,3)
\]

\[
\begin{array}{ccc}
5 & 5 & 5 \\
5 & 5 & 5 \\
5 & 5 & 5 \\
\end{array}
\]

To turn \(A\) into a 3-by-3-by-3-by-2, four-dimensional array, enter

\[
\begin{align*}
A(:,:,1,2) &= \begin{bmatrix} 1 & 2 & 3; & 4 & 5 & 6; & 7 & 8 & 9 \end{bmatrix}; \\
A(:,:,2,2) &= \begin{bmatrix} 9 & 8 & 7; & 6 & 5 & 4; & 3 & 2 & 1 \end{bmatrix}; \\
A(:,:,3,2) &= \begin{bmatrix} 1 & 0 & 1; & 1 & 1 & 0; & 0 & 1 & 1 \end{bmatrix};
\end{align*}
\]

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

**Generating Arrays Using 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,

\[
B = \text{randn}(4, 3, 2)
\]

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.

\[
B = \text{repmat}(5, [3 4 2])
\]

\[
\begin{array}{ccc}
5 & 5 & 5 \\
5 & 5 & 5 \\
5 & 5 & 5 \\
\end{array}
\]
Multidimensional Arrays

**B(:,:,2) =**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**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.

---

**Using the cat Function to Build Multidimensional Arrays**

The `cat` function is a simple way to build multidimensional arrays. It concatenates a list of arrays along a specified dimension,

\[
B = \text{cat}(\text{dim}, A1, A2, \ldots)
\]

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`:

\[
B = \text{cat}(3, \begin{bmatrix} 2 & 8; & 0 & 5 \end{bmatrix}, \begin{bmatrix} 1 & 3; & 7 & 9 \end{bmatrix})
\]

\[
B(:, :, 1) =
\begin{bmatrix}
2 & 8 \\
0 & 5
\end{bmatrix}
\]

\[
B(:, :, 2) =
\begin{bmatrix}
1 & 3 \\
7 & 9
\end{bmatrix}
\]
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.

```matlab
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

```matlab
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

```matlab
C(1, 2, 1, 2)
```
Getting Information About Multidimensional Arrays

You can use MATLAB functions and commands to get information about multidimensional arrays you have created.

<table>
<thead>
<tr>
<th>Information</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array size</td>
<td>size</td>
<td>size(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ans =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rows 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>columns 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dim 3 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dim 4 2</td>
</tr>
<tr>
<td>Array dimensions</td>
<td>ndims</td>
<td>ndims(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ans = 4</td>
</tr>
<tr>
<td>Array storage and format</td>
<td>whos</td>
<td>whos</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size 2x2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class double array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size 2x2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class double array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size 4-D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class double array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size 4-D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes 192</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class double array</td>
</tr>
<tr>
<td>Grand total is 48 elements using 384 bytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Working with Multidimensional Arrays

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well. This section describes how to apply basic indexing and reshaping techniques to multidimensional arrays.

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

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

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. 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

\[
\text{nddata(2, [1 3 4], 3)}
\]

The Colon and Multidimensional Array Indexing

MATLAB's 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,

\[
C = \text{zeros}(4, 4)
\]

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

\[
C(2:3, 2:3) = \text{nddata(2:3, 1:2, 2)}
\]

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**

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

\[ B = \text{reshape}(A, [s_1 \ s_2 \ s_3 \ldots]) \]

\( s_1, s_2, \) 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).

<table>
<thead>
<tr>
<th>M</th>
<th>reshape(M, [6 5])</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 7 8 5 2</td>
<td></td>
</tr>
<tr>
<td>3 5 8 5 1</td>
<td></td>
</tr>
<tr>
<td>6 9 4 3 3</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td></td>
</tr>
<tr>
<td>9 0 6 3 7</td>
<td></td>
</tr>
<tr>
<td>8 1 5 0 2</td>
<td></td>
</tr>
<tr>
<td>1 3 5 7 5</td>
<td></td>
</tr>
<tr>
<td>9 6 7 5 5</td>
<td></td>
</tr>
<tr>
<td>8 5 2 9 3</td>
<td></td>
</tr>
<tr>
<td>2 4 9 8 2</td>
<td></td>
</tr>
<tr>
<td>0 3 3 8 1</td>
<td></td>
</tr>
<tr>
<td>1 0 6 4 3</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>C</th>
<th>reshape(C, [6 2])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>1 6 3 8 2 9 4 11 5 10 7 12</td>
</tr>
</tbody>
</table>

Here are several new arrays from reshaping `nddata`:

\[
\begin{align*}
B &= \text{reshape}(\text{nddata}, [6 25]) \\
C &= \text{reshape}(\text{nddata}, [5 3 10]) \\
D &= \text{reshape}(\text{nddata}, [5 3 2 5])
\end{align*}
\]

**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.

\[
\begin{align*}
B &= \text{repmat}(5, [2 3 1 4]) \\
\text{size}(B) &= \\
\text{ans} &= 2 3 1 4
\end{align*}
\]

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

\[
\begin{align*}
C &= \text{squeeze}(B) \\
\text{size}(C) &= \\
\text{ans} &= 2 3 4
\end{align*}
\]
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.

\[
B = \text{permute}(A, \text{dims});
\]

where `\text{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.

<table>
<thead>
<tr>
<th>B = <code>permute(A,[2 1 3])</code></th>
<th>C = <code>permute(A,[3 2 1])</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>B(:,:,1) =</td>
<td>C(:,:,1) =</td>
</tr>
<tr>
<td>1  4  7</td>
<td>1  2  3</td>
</tr>
<tr>
<td>2  5  8</td>
<td>0  5  4</td>
</tr>
<tr>
<td>3  6  9</td>
<td>4  5  6</td>
</tr>
<tr>
<td>B(:,:,2) =</td>
<td>C(:,:,2) =</td>
</tr>
<tr>
<td>0  2  9</td>
<td>2  7  6</td>
</tr>
<tr>
<td>5  7  3</td>
<td></td>
</tr>
<tr>
<td>4  6  1</td>
<td></td>
</tr>
</tbody>
</table>

Row and column subscripts are reversed (page-by-page transposition).

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:

$$B = \text{permute}(A, [2, 4, 3, 1])$$

Inverse Permutation

The \text{i} \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e} function is the inverse of \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e}. Given an input array \(A\) and a vector of dimensions \(v\), \text{i} \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e} produces an array \(B\) such that \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e}(B, v)\) returns \(A\).

For example, these statements create an array \(E\) that is equal to the input array \(C\).

\[
D = \text{i} \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e}(C, [1, 4, 2, 3]); \\
E = \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e}(D, [1, 4, 2, 3])
\]

You can obtain the original array after permuting it by calling \text{i} \text{p} \text{e} \text{r} \text{m} \text{u} \text{t} \text{e} with the same vector of dimensions.
Computation with Multidimensional Arrays

Many of MATLAB’s 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.

Functions that Operate 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 three.

**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.

Functions that Operate 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.
Functions that Operate 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 operations `∗`, `^`, `\`, 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`:

\[
A = \text{cat}(3, [1 2 3; 9 8 7; 4 6 5], [0 3 2; 8 8 4; 5 3 5], [6 4 7; 6 8 5; \ldots 5 4 3])
\]

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 =

-2.6260
12.9129
2.7131
```

**NOTE** If the first 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`. 

---

12-16
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:

```
red_intensity
Page 1 - red intensity
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 1.000 0.867 0.051 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
```

```
green_intensity
Page 2 - 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 ...
```

```
blue_intensity
Page 3 - blue intensity
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 ...
```

To access an entire plane of the image, use:

```
red_plane = RGB(:, :, 1)
```
To access a subimage, use

\[
\text{subimage} = \text{RGB}(20:40, 50:85, :) ;
\]

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

\[
\text{mean} ( \text{mean} ( \text{mean} ( \text{TEMP} ) ) )
\]

To obtain a vector of the “middle” values in the room-element (2,2) on each page—use

\[
B = \text{TEMP}(2, 2, :);
\]
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`:

```matlab
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 this:
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:

```matlab
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).

\[ \text{sum}(\text{patient}(1, 1, 2).\text{test}) \]

Similarly, add all the billing fields in the patient array.

\[ \text{total} = \text{sum}([\text{patient}.\text{billing}]); \]
Structures and Cell Arrays

Structures

Building Structure Arrays ........................................... 13-3
Accessing Data in Structure Arrays ................................. 13-3
Using the size Function with Structure Arrays ............. 13-6
Adding Fields to Structures ........................................... 13-8
Deleting Fields from Structures .................................... 13-8
Applying Functions and Operators ................................. 13-8
Writing Functions to Operate on Structures ..................... 13-9
Organizing Data in Structure Arrays ............................. 13-10
Nesting Structures ....................................................... 13-15

Cell Arrays

Creating Cell Arrays ................................................... 13-18
Obtaining Data from Cell Arrays .................................. 13-18
Deleting Cells ............................................................. 13-23
Reshaping Cell Arrays .................................................. 13-24
Replacing Lists of Variables with Cell Arrays .................. 13-24
Applying Functions and Operators ............................... 13-26
Organizing Data in Cell Arrays ..................................... 13-27
Nesting Cell Arrays ..................................................... 13-28
Converting Between Cell and Numeric Arrays ................ 13-30
Cell Arrays of Structures .............................................. 13-31
Cell arrays are a special class of MATLAB arrays. Their elements consist of bins, or cells, that themselves contain MATLAB arrays. Cell arrays allow you to store dissimilar classes of arrays in the same array and to group-related data sets that have varying dimensions. Access is via normal matrix indexing operations.

Structures are another class of MATLAB arrays that allow you to store dissimilar arrays together. Structures differ from cell arrays in that they are referenced using named fields.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure functions</td>
<td>struct</td>
<td>Create or convert to structure array.</td>
</tr>
<tr>
<td></td>
<td>fieldnames</td>
<td>Get structure field names.</td>
</tr>
<tr>
<td></td>
<td>getfield</td>
<td>Get structure field contents.</td>
</tr>
<tr>
<td></td>
<td>setfield</td>
<td>Set structure field contents.</td>
</tr>
<tr>
<td></td>
<td>rmfield</td>
<td>Remove structure field.</td>
</tr>
<tr>
<td></td>
<td>isfield</td>
<td>True if field is in structure array.</td>
</tr>
<tr>
<td></td>
<td>isstruct</td>
<td>True for structures.</td>
</tr>
<tr>
<td>Cell array functions</td>
<td>cell</td>
<td>Create cell array.</td>
</tr>
<tr>
<td></td>
<td>celldisp</td>
<td>Display cell array contents.</td>
</tr>
<tr>
<td></td>
<td>cellplot</td>
<td>Display graphical depiction of cell array.</td>
</tr>
<tr>
<td></td>
<td>num2cell</td>
<td>Convert numeric array into cell array.</td>
</tr>
<tr>
<td></td>
<td>deal</td>
<td>Deal inputs to outputs.</td>
</tr>
<tr>
<td></td>
<td>cell2struct</td>
<td>Convert cell array into structure array.</td>
</tr>
<tr>
<td></td>
<td>struct2cell</td>
<td>Convert structure array into cell array.</td>
</tr>
<tr>
<td></td>
<td>iscell</td>
<td>True for cell array.</td>
</tr>
</tbody>
</table>
Structures

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 Chapter 12.

**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.

```matlab
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering

```matlab
patient
```

at the command line results in

```matlab
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.

```matlab
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:

```matlab
patient
```

```matlab
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

The `struct` function lets you preallocate an array of structures. Its basic form is

```matlab
str_array = struct(fields)
```

`fields` is a padded string array or cell array containing the field names for the structures. With this syntax, `struct` initializes every field in the array to an empty matrix. You can also use the syntax:

```matlab
str_array = struct('field1','val1','field2','val2', ...)
```

where the arguments are field names and their values. For example, you can use `struct` to preallocate a 1-by-2 `patient` array:

```matlab
patient = struct('name','John Doe','billing',127.00,...
                 'test',[79 75 73; 180 178 177.5; 220 210 205]);
```

`struct` initializes every structure with the specified field values. For example, all the `name` fields in the array created above contain the string `'John Doe'`, all the `billing` fields contain the value `127.00`, and so on. You can then modify field values or expand the array with assignment statements as shown earlier.
Accessing Data in Structure Arrays

Using structure array indexing, you can get the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. For the examples in this section, consider the structure:

To access an entire field, include a period (.) after the structure array name, followed by the field name:

```matlab
str = patient(2).name
```

```matlab
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:

```matlab
n = patient(3).test(2,2)
```

```matlab
n =

153
```

Use the same notations to assign values to structure fields, for example:

```matlab
patient(3).test(2,2) = 7;
```
You cannot obtain field values for multiple structures in an array at one time; that is, you have to obtain field values for each element in the structure array individually. For example, to display the value of every name field use a loop:

```matlab
for i = 1:length(patient)
    disp(patient(i).name);
end
```

Access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-1 structure array for which the single element is the second structure of the patient array:

```matlab
B = patient(2)
```

**Accessing Field Values Using setfield and getfield**

Direct indexing is usually the most efficient way to assign or retrieve field values. If, however, you only know the field name as a string—for example, if you have used the fieldnames function to obtain the field name within an M-file—you can use the setfield and getfield functions to do the same thing.

`getfield` obtains a value or values from a field or field element:

```matlab
f = getfield(array,{array_index},'field',{field_index})
```

where the `field_index` is optional, and `array_index` is optional for a 1-by-1 structure array. The function syntax corresponds to

```matlab
f = array(array_index).field(field_index);
```

For example, to access the name field in the second structure of the patient array, use

```matlab
str = getfield(patient, {2}, 'name')
```

Similarly, `setfield` lets you assign values to fields using the syntax

```matlab
f = setfield(array, {array_index}, 'field d', {field d_index}, value)
```
Using the size Function with 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 3-by-1 structure array `patient`, `size(patient)` returns the vector `[3 1]`. The statement `size(patient(2,1).name)` returns the length of the `name` string for element `(2, 1)` 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

```matlab
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.

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

```matlab
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

```matlab
patient2 = 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, to find the mean across the rows of the `test` array in `patient(2)`:  

```matlab
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

```matlab
    total = 0;
    for j = 1:length(patient)
        total = total + patient(j).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

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

This is equivalent to using the comma-separated list:

```matlab
    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`.

```matlab
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`:

```matlab
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:

<table>
<thead>
<tr>
<th>Blue intensity values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.689 0.706 0.118 0.884 ...</td>
</tr>
<tr>
<td>0.535 0.532 0.653 0.925 ...</td>
</tr>
<tr>
<td>0.314 0.265 0.159 0.101 ...</td>
</tr>
<tr>
<td>0.553 0.633 0.528 0.493 ...</td>
</tr>
<tr>
<td>0.441 0.465 0.512 0.512 ...</td>
</tr>
<tr>
<td>0.398 0.401 0.421 0.398 ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Green intensity values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.342 0.647 0.515 0.816 ...</td>
</tr>
<tr>
<td>0.111 0.300 0.205 0.526 ...</td>
</tr>
<tr>
<td>0.523 0.428 0.712 0.929 ...</td>
</tr>
<tr>
<td>0.214 0.604 0.918 0.344 ...</td>
</tr>
<tr>
<td>0.100 0.121 0.113 0.126 ...</td>
</tr>
<tr>
<td>0.288 0.187 0.204 0.175 ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RED intensity values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.112 0.986 0.234 0.432 ...</td>
</tr>
<tr>
<td>0.765 0.128 0.863 0.521 ...</td>
</tr>
<tr>
<td>1.000 0.985 0.761 0.698 ...</td>
</tr>
<tr>
<td>0.455 0.783 0.224 0.395 ...</td>
</tr>
<tr>
<td>0.021 0.500 0.311 0.123 ...</td>
</tr>
<tr>
<td>1.000 1.000 0.867 0.051 ...</td>
</tr>
<tr>
<td>1.000 0.945 0.998 0.893 ...</td>
</tr>
<tr>
<td>0.990 0.941 1.000 0.876 ...</td>
</tr>
<tr>
<td>0.902 0.867 0.834 0.798 ...</td>
</tr>
</tbody>
</table>

There are at least two ways you can organize such data into a structure array:
Plane Organization

In case 1 above, each field of the structure is an entire plane of the image. You can create this structure using

\[
\begin{align*}
A.r &= \text{RED;} \\
A.g &= \text{GREEN;} \\
A.b &= \text{BLUE;} \\
\end{align*}
\]

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

\[
\text{red\_plane} = A.\text{red};
\]

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:

```matlab
red_sub = A.r(2:12,13:30);
grn_sub = A.g(2:12,13:30);
blu_sub = A.b(2:12,13:30);
```

**Element-by-Element Organization**

Case 2 has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```matlab
for i = 1:size(RED,1)
    for j = 1:size(RED,2)
        B(i,j).r = RED(i,j);
        B(i,j).g = GREEN(i,j);
        B(i,j).b = BLUE(i,j);
    end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```matlab
Bsub = B(1:10,1:10);
```

To access an entire plane of the image using the element-by-element method, however, requires a loop:

```matlab
red_plane = zeros(128,128);
for i = 1:(128*128)
    red_plane(i) = B(i).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:
  ```matlab
  avg = mean(A.amount);
  ```

  Using element-by-element organization:
  ```matlab
  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:
```matlab
function client(name, address)
disp(name)
disp(address)
```

Using element-by-element organization, pass an entire structure:
```matlab
function client(B)
disp(B)
```

To call the client function,

Using plane organization:
```matlab
client(A.name(2,:), A.address(2,:))
```

Using element-by-element organization:
```matlab
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-2 structure array:
```matlab
A = struct([1 2], 'data',[3 4 7; 8 0 1], 'nest',...
    struct([1 1], 'testnum', 'Test 1', 'xdata',[4 2 8],...
    'ydata',[7 1 6]))
```
A(1) has the desired values because of the `struct` call. Modify A(2):

```
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];
```

As with flat structure arrays (structure arrays that are only one layer deep), you can build nested structure arrays using direct assignment statements. This code produces the same result as the example on the previous page:

```
A(1).data = [3 4 7; 8 0 1];
A(1).nest.testnum = 'Test 1';
A(1).nest.xdata = [4 2 8];
A(1).nest.ydata = [7 1 6];
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 two 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).
Cell Arrays

A cell array is a MATLAB array for which the elements are cells, containers that can hold other MATLAB arrays. For example, one cell of a cell array might contain a real matrix, another an array of text strings, and another a vector of complex values.

You can build cell arrays of any valid size or shape, including multidimensional structure arrays.

**NOTE** The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see Chapter 12.

### Creating Cell Arrays

You can create cell arrays by:

- Using assignment statements
- Preallocating the array using the `cells` function, then assigning data to cells
Using Assignment Statements

You can build a cell array by assigning data to individual cells, one cell at a time. MATLAB automatically builds the array as you go along. There are two ways to assign data to cells:

• Cell indexing.
  Enclose the cell subscripts in parentheses using standard array notation. Enclose the cell contents on the right side of the assignment statement in curly braces, “{}.” For example, create a 2-by-2 cell array A:
  
  \[
  \begin{align*}
  A(1, 1) &= \{[1 \ 4 \ 3; \ 0 \ 5 \ 8; \ 7 \ 2 \ 9]\}; \\
  A(1, 2) &= \{'Anne Smith'\}; \\
  A(2, 1) &= \{3+7i\}; \\
  A(2, 2) &= \{-\pi : \pi / 10 : \pi\}
  \end{align*}
  \]

**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.

• Content indexing.
  Enclose the cell subscripts in curly braces using standard array notation. Specify the cell contents on the right side of the assignment statement:
  
  \[
  \begin{align*}
  A\{1, 1\} &= \{[1 \ 4 \ 3; \ 0 \ 5 \ 8; \ 7 \ 2 \ 9]\}; \\
  A\{1, 2\} &= \{'Anne Smith'\}; \\
  A\{2, 1\} &= \{3+7i\}; \\
  A\{2, 2\} &= \{-\pi : \pi / 10 : \pi\}
  \end{align*}
  \]

The various examples in this guide do not use one syntax throughout, but attempt to show representative usage of cell and content addressing. You can use the two forms interchangeably.
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.

MATLAB displays the cell array \( A \) in a condensed form

\[
A = \\
\begin{bmatrix}
3 \times 3 \text{ double} & \text{'Anne Smith'} \\
3 \times 0.0000 + 7 \times 0.0000i & 1 \times 21 \text{ double}
\end{bmatrix}
\]

To display the full cell contents, use the `celldisp` function. For a high-level graphical display of cell architecture, use `cellplot`.

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 Arrays

Cell Array Syntax: Using Braces
The curly braces, “{}”, are cell array constructors, just as square brackets are numeric array constructors. Curly braces behave similarly to square brackets, except that you can nest curly braces to denote nesting of cells (see page 13-28 for details).

Curly braces use commas or spaces to indicate column breaks and semicolons to indicate row breaks between cells. For example,

\[ C = \{[1 \ 2], \ [3 \ 4]; \ [5 \ 6], \ [7 \ 8]\} \]

results in

<table>
<thead>
<tr>
<th>cell 1,1</th>
<th>cell 1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 \ 2]</td>
<td>[3 \ 4]</td>
</tr>
<tr>
<td>cell 2,1</td>
<td>cell 2,2</td>
</tr>
<tr>
<td>[5 \ 6]</td>
<td>[7 \ 8]</td>
</tr>
</tbody>
</table>

Use square brackets to concatenate cell arrays, just as you do for numeric arrays.

Preallocating Cell Arrays with the cell Function
The cell function allows you to preallocate empty cell arrays of the specified size. For example, to create an empty 2-by-3 cell array,

\[ B = \text{cell}(2, 3) \]

Use assignment statements to fill the cells of B; for example,

\[ B(1, 3) = \{1:3\}; \]

Obtaining Data from Cell Arrays
You can obtain data from cell arrays and store the result as either a standard array or a new cell array. This section discusses:

- Accessing cell contents using content indexing
- Accessing a subset of cells using cell indexing
Accessing Cell Contents Using Content Indexing

You can use content indexing on the right side of an assignment to access some or all of the data in a single cell. Specify the variable to receive the cell contents on the left side of the assignment. Enclose the cell index expression on the right side of the assignment in curly braces. This indicates that you are assigning cell contents, not the cells themselves.

Consider the 2-by-2 cell array N.

\[
N{1, 1} = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}; \\
N{1, 2} = 'Name'; \\
N{2, 1} = 2-4i; \\
N{2, 2} = 7;
\]

You can obtain the string in \( N{1, 2} \) using

\[
c = N{1, 2}
\]

\[
c = Name
\]

**NOTE** In assignments, you can use content indexing to access only a single cell, not a subset of cells. For example, the statements \( A{1, :} = \text{value} \) 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 “section Replacing Lists of Variables with Cell Arrays” on page 13-24 for details.

To obtain subsets of a cell’s contents, concatenate indexing expressions. For example, to obtain element \((2, 2)\) of the array in cell \( N{1, 1} \), use:

\[
d = N{1, 1}(2, 2)
\]

\[
d = 5
\]
Accessing a Subset of Cells Using Cell Indexing

Use cell indexing to assign any set of cells to another variable, creating a new cell array. Use the colon operator to access subsets of cells within a cell array.

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.

\[
A(\text{cell}_\text{subscripts}) = []
\]

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.

```matlab
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:

\[
\begin{align*}
C(1) & = \{[1 \ 2 \ 3]\}; \\
C(2) & = \{[1 \ 0 \ 1]\}; \\
C(3) & = \{1:10\}; \\
C(4) & = \{[9 \ 8 \ 7]\}; \\
C(5) & = \{3\};
\end{align*}
\]

To convolve the vectors in C(1) and C(2) using \texttt{conv},

\[
d = \text{conv}(C{1:2})
\]

\[
d = \\
\begin{bmatrix}
1 & 2 & 4 & 2 & 3
\end{bmatrix}
\]

Display vectors two, three, and and four with

\[
C{2:4}
\]

\[
\begin{align*}
\text{ans} & = \\
& = \\
& = \\
& = \\
& = \\
\end{align*}
\]

\[
\begin{align*}
\text{ans} & = \\
& = \\
& = \\
& = \\
& = \\
\end{align*}
\]

\[
\begin{align*}
\text{ans} & = \\
& = \\
& = \\
& = \\
& = \\
\end{align*}
\]

Similarly, you can create a new numeric array using the statement

\[
B = [C(1); \ C(2); \ C(4)]
\]

\[
B = \\
\begin{bmatrix}
1 & 2 & 3 \\
1 & 0 & 1 \\
9 & 8 & 7
\end{bmatrix}
\]
You can also use content indexing on the left side of an assignment to create a new cell array, each cell of which represents a separate output argument:

\[ D[1: 2] = \text{eig}(B) \]

\[ D = \]

\[
\begin{bmatrix}
3x3 \text{ double} & 3x3 \text{ double}
\end{bmatrix}
\]

You can display the actual eigenvalues and eigenvectors 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 Chapter 10 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.

\[
A[1, 1] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix};
A[1, 2] = \text{randn}(3, 3);
A[1, 3] = 1:5;
B = \text{sum}(A[1, 1])
\]

\[ B = \]

\[
\begin{bmatrix}
4 & 6
\end{bmatrix}
\]

To apply a function to several cells of a non-nested cell array, use a loop:

```matlab
for i = 1:length(A)
    M{i} = sum(A{i});
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:

For cell array TEST:

```
[newdata, name] = deal (TEST{1:2})
```

For 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 \( B \):

\[
[B{1:2}] = \text{normest}(A)
\]

\[
B = \\
\begin{bmatrix}
8.8826 & 4
\end{bmatrix}
\]

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 `cel ls` 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:

```matlab
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 []}}} 
```

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.
   \[ A = \text{cell}(1, 2) \]

2. Create a 2-by-2 cell array inside \( A(1, 2) \).
   \[ A(1, 2) = \{ \text{cell}(2, 2) \} \]

3. Fill \( A \), including the nested array, using assignments.
   \[
   \begin{align*}
   A(1, 1) &= \{ \text{magic}(5) \}; \\
   A(1, 2)(1, 1) &= \{ [5 \ 2 \ 8; \ 7 \ 3 \ 0; \ 6 \ 7 \ 3] \}; \\
   A(1, 2)(1, 2) &= \{ \text{Test} \ 1' \}; \\
   A(1, 2)(2, 1) &= \{ [2-4i \ 5+7i] \}; \\
   A(1, 2)(2, 2) &= \{ \text{cell}(1, 2) \} \\
   A(1, 2){2, 2}(1) &= \{ 17 \};
   \end{align*}
   \]

   Note the use of curly braces until the final level of nested subscripts. This is 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.

<table>
<thead>
<tr>
<th>cell 1,1</th>
<th>cell 1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 24 1  8 15</td>
<td></td>
</tr>
<tr>
<td>23 5  7 14 16</td>
<td></td>
</tr>
<tr>
<td>4 6  13 20 22</td>
<td></td>
</tr>
<tr>
<td>10 12 19 21 3</td>
<td></td>
</tr>
<tr>
<td>11 18 25 2  9</td>
<td></td>
</tr>
<tr>
<td>5  2  8</td>
<td></td>
</tr>
<tr>
<td>7  3  0</td>
<td></td>
</tr>
<tr>
<td>6  7  3</td>
<td></td>
</tr>
<tr>
<td>'Test 1'</td>
<td></td>
</tr>
<tr>
<td>[2-4i 5+7i]</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td></td>
</tr>
</tbody>
</table>

- 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:

```matlab
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:

```matlab
for k = 1:4
    for i = 1:2
        for j = 1:2
            NUM(i,j,k) = F{k}(i,j);
        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.

```
c_str = cell(1, 2)
c_str{1}.label = '12/2/94 – 12/5/94';
c_str{1}.obs = [47 52 55 48; 17 22 35 11];
c_str{2}.xdata = [-0.03 0.41 1.98 2.12 17.11];
c_str{2}.ydata = [-3 5 18 0 9];
c_str{2}.zdata = [0.6 0.8 1 2.2 3.4];
```

Cell 1 of the `c_str` 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

```
cell_array{index}.field
```

For example, to access the `label` field of the structure in cell 1, use `c_str{1}.label`.
Classes and Objects

Classes and Objects: An Overview .................................. 14-2
Class Directories ......................................................... 14-3
Data Structures ............................................................. 14-4
Constructors ............................................................... 14-4
class and isa functions ............................................... 14-6
Converter Functions ..................................................... 14-6
Displayed Output ......................................................... 14-8

Overloading ................................................................. 14-10
Overloading Arithmetic ............................................... 14-10
Overloading Operators ............................................... 14-11
Overloading Functions ............................................... 14-13

Object Precedence ........................................................ 14-16
Objects and Arrays ..................................................... 14-16
Object Indexing Within Methods ..................................... 14-19

Inheritance ................................................................. 14-22
Simple Inheritance ..................................................... 14-22
Multiple Inheritance .................................................. 14-25
Aggregation ............................................................... 14-27
How MATLAB Calls Methods ....................................... 14-27
Private Methods and Private Functions ............................. 14-27
Debugging Object Methods ............................................ 14-28
Classes and Objects

This chapter describes how to add new data types to MATLAB by creating classes. It also explains how to create and manipulate objects, which are instances of MATLAB classes.

Classes and Objects: An Overview

Classes and objects allow you to add new data types and new operations to MATLAB. The class of a variable describes the structure of the variable and indicates the kinds of operations and functions that can apply to the variable. An object is a variable or an instance of a particular class. The phrase “object-oriented programming” describes an approach to writing programs that emphasizes the use of classes and objects.

MATLAB has five built-in classes.

<table>
<thead>
<tr>
<th>Table 1-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>double</strong></td>
</tr>
<tr>
<td><strong>sparse</strong></td>
</tr>
<tr>
<td><strong>char</strong></td>
</tr>
<tr>
<td><strong>struct</strong></td>
</tr>
<tr>
<td><strong>cell</strong></td>
</tr>
</tbody>
</table>

Various MATLAB toolboxes provide additional class definitions. For example, MATLAB provides the `inline` class, which is a way of generating simple, “one line” function definitions for use with quadrature, differential equation, and root-finding routines. The Symbolic Math Toolbox is based upon the `sym` class, which manipulates symbolic variables and matrices. The Control System Toolbox defines the `lti` class and three subclasses, which provide analysis of linear, time-invariant systems.

You can add classes to your own MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a directory of M-files that operate on the object. These M-files are known as the methods for the class. The class directory may 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.

The MATLAB language does not have declarations. For example, the statement

```matlab
A = zeros(10, 10)
```
creates a traditional MATLAB matrix, which is of class `double`. No declaration of `A` is required. Similarly,

```matlab
s = 'Hello world'
```
creates an instance of class `char` without any declaration of `s`.

The same considerations apply to the new classes you define. There are no declarations. The objects are created dynamically, by invoking the constructor for the class.

An example used throughout 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`. With this class, an object representing the polynomial

\[ p(x) = x^3 - 2x - 5 \]

is created by calling the `polynom` constructor with a vector of coefficients

```matlab
p = polynom([1 0 -2 -5])
```

## Class Directories

The M-files defining the methods for a class are collected together in directories. The directory names are formed with the class name preceded by the character `@`. (On VAX/VMS systems the character `$` is used instead of the `@`.) For example, the M-files defining the polynomial class would be located in directories with the name `@polynom`.

The methods directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For example, `@inline` is a subdirectory of `toolbox/MATLAB/funfun` and `@sym` is a subdirectory of `toolbox/symbolic`. The new `@polynom` directory would be a subdirectory of MATLAB’s working directory or your own personal directory that has been added to the search path.

The union of the files in all these directories forms the methods for the class.
Data Structures

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.

For the polynomial example, we have chosen to represent a polynomial by a row vector containing the coefficients of powers of the variable, in decreasing order. So 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.

This is not the only way to represent a polynomial. The coefficients could be ordered by increasing powers, or be stored in a column vector. It is even possible to represent a polynomial, up to a single scalar multiplier, by specifying its zeros. The choice among these alternate representations of a simple object like a polynomial is not particularly difficult or important, but for more sophisticated objects, the design of the appropriate data structure can be crucial. The data structure chosen for sparse matrices, for example, has significant implications for the execution time of sparse operations.

Constructors

The methods 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 class, and of the directory without the @prefix. The constructor creates the objects by initializing the data structure and assigning the class tag.
Here is the polynomial constructor, \texttt{polynom}.

\begin{verbatim}
function p = polynom(a)
    %POLYNOM Polynomial class constructor.
    % p = POLYNOM(v) creates a polynomial object from the 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

    It is possible that MATLAB will call the constructor with no arguments. In this
    case, the constructor should produce a template for the object, usually with
    empty fields. It is also possible that the constructor will be called with an input
    argument that is already in the class. In this case, the constructor usually
    returns the input. The function \texttt{isa} (pronounced “is a”) checks for this
    situation. If the input argument exists and is not a \texttt{polynom} it is reshaped to
    be a row vector and assigned to the .c field of the result. Finally, the \texttt{class}
    function is used to assign a tag to the result that identifies it as a \texttt{polynom}.

    An example of the use of the \texttt{polynom} constructor is the statement
    \begin{verbatim}
    p = polynom([1 0 -2 -5])
    \end{verbatim}

    This creates a polynomial with the specified coefficients. The output resulting
    from this example, as well as other operations on \texttt{p}, is discussed in later
    sections.

    In general, constructor functions should follow this outline.
    \begin{itemize}
    \item If there are no arguments, return a template object.
    \item If the argument is already in the class, return it.
    \item Convert the input to the desired form.
    \item Assign the various fields in the structure.
    \item Use \texttt{class} to assign the appropriate tag.
    \end{itemize}
\end{verbatim}
class and isa functions

The class and isa functions used in the constructor can also be used outside of the methods directory. The expression

```matlab
isa(a, 'class_name')
```

checks if `a` is an object of the specified class. For example, each of the following expressions is true.

```matlab
isa(pi, 'double')
isa('hello', 'char')
isa(p, 'polynom')
```

Outside of the methods, class takes only one argument. The expression

```matlab
class(a)
```

returns a string containing the class name of `a`. For example

```matlab
class(pi),
class('hello'),
class(p)
```

return

```matlab
'double',
'char',
'polynom'
```

Converter Functions

A converter function call is of the form

```matlab
b = class_name(a)
```

where `a` is an object of a class other than `class_name`. In this situation, MATLAB looks for a method called `class_name` in the class directory for object `a`. Such a method converts an object of one class to an object of another class. If the input object is already of type `class_name`, then MATLAB calls the constructor—which typically returns its input.

Two of the most important converter functions are `double` and `char`. Conversion to `double` produces MATLAB's traditional matrix, although this
may not be appropriate for some classes. Conversion to \texttt{char} is useful for producing printed output.

The converter to \texttt{double} for the polynomial class is a very simple M-file, \texttt{@polynom/double.m} which merely retrieves the coefficient vector.

\begin{verbatim}
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;
\end{verbatim}

On the sample polynomial,

\begin{verbatim}
double(p)
\end{verbatim}

returns

\begin{verbatim}
ans =
   1    0   -2   -5
\end{verbatim}

The conversion to \texttt{char} is a key method because it produces a character string involving the powers of an independent variable, \texttt{x}. In fact, once \texttt{x} has been specified, the string is a syntactically correct MATLAB expression. Here is \texttt{@polynom/char.m}.

\begin{verbatim}
function s = char(p)
    \% POLYNOM/CHAR  CHAR(p) is the string representation of p.
    \% c = p.c;
    if all(c == 0)
        s = '0';
    else
        d = length(p.c)-1;
        s = ['
        for a = c;
            if a ~= 0;
                if ~isempty(s)
                    if a > 0
                        s = [s ' + '];
                    else
                        s = [s ' - '];
                    end
                    a = -a;
                end
            end
        end
\end{verbatim}
Classes and Objects

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
d = d - 1;
end
end

On the sample polynomial,
char(p)
produces the result
ans =
x^3 - 2*x - 5

Displayed Output
A method named display is called whenever an object results from a MATLAB statement that is not terminated by a semicolon. In many classes, display can simply print the variable name and then use the char converter to print the contents or value of the variable. Here is @polynom DISPLAY.m. The body of this function can be used without change in other methods' directories.
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom

disp(' ');
disp([inputname(1), ' = '])
disp(' ');
disp([' ' char(p)])
disp(' ');

When working with polynomials, it is useful to have an object, \( x \), which
represents the polynomial's independent variable, \( x \). This is accomplished with
the statement

\[
x = \text{polynom}([1 0])
\]

Since the statement is not terminated with a semicolon, the resulting output is

\[
x = x
\]
Overloading

In many cases, you may want to change the behavior of MATLAB’s operators and functions for object arguments. You can accomplish this by overloading the relevant functions. Overloading enables a function to handle different types and numbers of input arguments.

Overloading Arithmetic

Each built-in MATLAB operator has an associated function name. 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 a polynom, the expression

\[
p + q
\]

generates a call to a function \(@polynom\) plus \(.m\) if it exists. Here is the M-file.

```matlab
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.

Here is another example, \(@polynom\) times \(.m\) which is called to compute the product \( p \times q \). The letter mat the beginning of the function name comes from the
fact that this is overloading MATLAB’s matrix multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```matlab
function r = mtimes(p, q)
% POLYNOM MTIMES   Implement p * q for polynomials.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c, q.c));
```

These two functions are used by the statements

```matlab
q = p + 1
r = p*q
```

to produce

```matlab
q = x^3 - 2*x - 4
r = x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

## Overloading Operators

The following table lists the function names for most of MATLAB’s operators.

<table>
<thead>
<tr>
<th>Operation</th>
<th>M-File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>plus(a, b)</td>
<td>Binary addition</td>
</tr>
<tr>
<td>a - b</td>
<td>minus(a, b)</td>
<td>Binary subtraction</td>
</tr>
<tr>
<td>-a</td>
<td>uminus(a)</td>
<td>Unary minus</td>
</tr>
<tr>
<td>+a</td>
<td>uplus(a)</td>
<td>Unary plus</td>
</tr>
<tr>
<td>a.*b</td>
<td>times(a, b)</td>
<td>Element-wise multiplication</td>
</tr>
<tr>
<td>a*b</td>
<td>mtimes(a, b)</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>a./b</td>
<td>rdivide(a, b)</td>
<td>Right element-wise division</td>
</tr>
<tr>
<td>a.</td>
<td>ldivide(a, b)</td>
<td>Left element-wise division</td>
</tr>
</tbody>
</table>
### Table 2-1:

<table>
<thead>
<tr>
<th>Operation</th>
<th>M-File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a/b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a\b</td>
<td>mldiv(a,b)</td>
<td>Matrix left division</td>
</tr>
<tr>
<td>a .^ b</td>
<td>power(a,b)</td>
<td>Element-wise power</td>
</tr>
<tr>
<td>a .^ b</td>
<td>mpower(a,b)</td>
<td>Matrix power</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>lt(a,b)</td>
<td>Less than</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>gt(a,b)</td>
<td>Greater than</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>le(a,b)</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>ge(a,b)</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>a == b</td>
<td>ne(a,b)</td>
<td>Not equal to</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>eq(a,b)</td>
<td>Equality</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>and(a,b)</td>
</tr>
<tr>
<td>~a</td>
<td></td>
<td>Logical NOT</td>
</tr>
<tr>
<td>a: d: b</td>
<td>colon(a,d,b)</td>
<td>Colon operator</td>
</tr>
<tr>
<td>a: b</td>
<td>colon(a,b)</td>
<td></td>
</tr>
<tr>
<td>a'</td>
<td>ctranspose(a)</td>
<td>Complex conjugate transpose</td>
</tr>
<tr>
<td>a.'</td>
<td>transpose(a)</td>
<td>Matrix transpose</td>
</tr>
<tr>
<td>command</td>
<td>window</td>
<td>out put</td>
</tr>
<tr>
<td>[ a b ]</td>
<td>horzcat(a,b,...)</td>
<td>Horizontal concatenation</td>
</tr>
<tr>
<td>[ a; b ]</td>
<td>vertcat(a,b,...)</td>
<td>Vertical concatenation</td>
</tr>
<tr>
<td>a(s1,s2,...sn)</td>
<td>subsref(a,s)</td>
<td>Subscripted reference</td>
</tr>
<tr>
<td>a(s1,...,sn) = b</td>
<td>subsasgn(a,s,b)</td>
<td>Subscripted assignment</td>
</tr>
<tr>
<td>b(a)</td>
<td>subindex(a,b)</td>
<td>Subscript index</td>
</tr>
</tbody>
</table>
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. Here are a few examples involving 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 polynomial object. In many cases, the overloading function can simply apply the original function to the coefficient field. For example, here is `@polynom/roots.m`.

```matlab
function r = roots(p)
    % POLYNOM/ROOTS. ROOTS(p) is a vector containing the roots of p.
    r = roots(p.c);
end
```

The statement

```matlab
roots(p)
```
results in

```matlab
ans =
  2.0946
-1.0473 + 1.1359i
-1.0473 - 1.1359i
```

The function `polyval` evaluates a polynomial at a given set of points. Here is `@polynom/polyval.m`. It uses nested multiplication, or Horner’s method.

```matlab
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
end
```

Both of these functions are used in an overloaded `plot` function. The domain of the independent variable is chosen to be slightly larger than an interval
containing all real roots. Then `polyval` is used to evaluate the polynomial at a few hundred points in the domain. Here is `@polynom/plot.m`:

```matlab
function plot(p)
% POLYNOM/plot  PLOT(p) plots the polynom p.
    r = max(abs(roots(p)));
    x = (-1.1:.01:1.1)*r;
    y = polyval(p,x);
    plot(x,y);
    title(char(p))
    grid on
```

Finally, here is `@polynom/diff.m`, which differentiates a polynomial by multiplying its coefficients by the appropriate indices.

```matlab
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));
```

The function call

```matlab
methods('polynom')
```

or its command form

```matlab
methods polynom
```

shows all the methods available for a particular class. For the `polynom` example, the output is

```
Methods for class polynom:
char    display    minus    mtimes    plus    polyval    roots
diff    double     mrdivide  plot      polynom   rem
```

Most of these methods can be exercised with the statement

```matlab
plot(diff(p*p + 10*p + 20*x) – 20)
```
The graph represents the function $6x^5 - 16x^3 + 8x$.
Object Precedence

Object precedence sets up a hierarchy between MATLAB objects. This allows you to control behavior for expressions like $a + b$ and $b + a$. Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. If you have established a precedence relationship, MATLAB calls the method for the class with the highest precedence. The functions `inferioro` and `superioro` should be invoked from the constructor to place the object into a hierarchy with other objects.

The `superioro` function places an object above other objects in the precedence hierarchy. For example, suppose that we want to add a `rational` class and that mixed expressions involving polynomials and rational functions are expected. This can be accomplished without making any changes to the `polynomial` methods. The constructor `@rational/rational.m` should include the statement

```matlab
superioro('polynomial')
```

Then expressions like `p+r`, `r+p`, `p*r` and `r*p` involving a polynomial `p` and a rational `r` will use the methods in `@rational`. Similarly, the `inferioro` function places the object lower in the precedence hierarchy than the specified classes. This means that the classes given as arguments to `inferioro` always have precedence over the object being created.

Objects and Arrays

You may recall the trigonometric identity

$$\cos 2\theta = 2\cos^2 \theta - 1$$

This is a special case of the fact that $\cos n\theta$ is a polynomial of degree $n$ in $\cos \theta$. These polynomials are known as the Chebyshev polynomials of the first kind.

$$T_n(x) = \cos n\theta$$

where $x = \cos \theta$

The trigonometric identity

$$\cos (n+1)\theta + \cos (n-1)\theta = 2\cos \theta \cos n\theta$$

leads to the fundamental recursion relation for the Chebyshev polynomials:

$$T_0(x) = 1$$
\[
T_1(x) = x
\]
\[
T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x), \quad n > 1
\]

What is \(T_{10}(x)\)? The Chebyshev polynomials form a sequence of polynomials. How can MATLAB and the polynomial object best be used to generate this sequence?

These questions lead to the more general question of the relationship between MATLAB's arrays and MATLAB's objects. There are at least three distinctly different ways to combine arrays and objects.

- An object's fields can be arrays.
- An object can be an array.
- An array's elements can be objects.

This leads to three possible approaches to generating a sequence of polynomials.

1. **The object's field is an array.** The polynomial object described earlier in this chapter already has a field that is an array, that is, the row vector containing the polynomial coefficients. The notion of “polynomial” could be generalized to have coefficients that are vectors themselves. Then the polynomial object would have a field, \(p.c\), consisting of a two-dimensional array or a cell array of coefficient vectors. All the methods for this generalized polynomial method would have to handle such coefficients. This is probably not the best design for a polynomial sequence object, although it is certainly a viable data structure for other objects.

2. **An object is an array.** In this case, the polynomial object would be based on a structure array. Within the methods, the polynomials would be denoted by \(p(k)\) and each polynomial would have its own coefficient field, \(p(k).c\). Each method would loop over the polynomials in the sequence. This would make implementation of the methods more complex, but use of the objects might be easier.

3. **An array of objects.** The polynomial object described earlier can be used without alteration to generate a sequence of polynomials stored in a cell array. This is probably the best way to generate the Chebyshev polynomials.
Recall that the independent variable $x$ can be thought of as the polynomial $1x + 0$ with coefficient vector $[1 \ 0]$. That is,

$$x = \text{polynom}([1 \ 0]);$$

This allows the first 10 Chebyshev polynomials to be generated and stored in a cell array with a few statements derived directly from the fundamental recursion relation:

```matlab
T{1} = 1;
T{2} = x;
for n = 2:10
    T{n+1} = 2*x*T{n} - T{n-1};
end
```

The polynomials generated are:

1
$x$
$2xx^2 - 1$
$4xx^3 - 3xx$
$8xx^4 - 8xx^2 + 1$
$16xx^5 - 20xx^3 + 5xx$
$32xx^6 - 48xx^4 + 18xx^2 - 1$
$64xx^7 - 112xx^5 + 56xx^3 - 7xx$
$128xx^8 - 256xx^6 + 160xx^4 - 32xx^2 + 1$
$256xx^9 - 576xx^7 + 432xx^5 - 120xx^3 + 9xx$
$512xx^{10} - 1280xx^8 + 1120xx^6 - 400xx^4 + 50xx^2 - 1$

Plotting the last one with

```matlab
plot(T{11})
```
shows the characteristic Chebyshev equal ripple property on the interval $-1 \leq x \leq 1$, as well as the rapid growth outside this interval.

Object Indexing Within Methods
This section discusses the relationship between indices and objects. For example, if $p$ is a polynomial, what is meant by the expression

$p(3)$

For the object as described so far, $p(3)$ produces an error. But, it could mean any of the following:

- Value of the polynomial at $x = 3$
- Third derivative
- Third polynomial in a sequence of polynomials
- Third coefficient in a single polynomial
- Coefficient of $x^3$
A complete design of a polynomial object might choose one of these interpretations. The functions discussed in this section would then be used to overload the interpretation of indexing operations.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. See Chapter 13 of this guide for details.

**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. The relevant expressions include

\[
\begin{align*}
A(1) \\
A(1) \\
A.field
\end{align*}
\]

Each of these results in a call to `subsref` of the form:

\[
B = \text{subsref}(A, S)
\]

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 the string `': '`.

For instance, the expression

\[
A(1:2,:)
\]

calls `subsref(A, S)`, where `S` is a 1-by-1 structure with

\[
\begin{align*}
S\text{.type} &= '( )' \\
S\text{.subs} &= \{1:2, ': '\}
\end{align*}
\]

Similarly, the expression

\[
A\{1:2\}
\]

goes through

\[
\begin{align*}
S\text{.type} &= '{ }' \\
S\text{.subs} &= \{1:2\}.
\end{align*}
\]
The expression
\[ A.\text{field} \]
calls \texttt{subsref(A, S)} where
\[ S.\text{type} = '.' \]
\[ S.\text{subs} = '\text{field}' \]

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases \texttt{length(S)} is the number of subscripting levels. For instance,
\[ A(1, 2).\text{name}(3:4) \]
calls \texttt{subsref(A, S)} where \( S \) is 3-by-1 structure array with the following values:
\[ S(1).\text{type} = '()' \]
\[ S(2).\text{type} = '.' \]
\[ S(3).\text{type} = '()' \]
\[ S(1).\text{subs} = '{1, 2}' \]
\[ S(2).\text{subs} = 'name' \]
\[ S(3).\text{subs} = '{3:4}' \]

\section*{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 \texttt{subsasgn} in these situations. The relevant statements include
\[ A(I) = B \]
\[ A(I) = B \]
\[ A.\text{field} = B \]

Each of these results in a call to \texttt{subsasgn} of the form:
\[ A = \texttt{subsasgn}(A, S, B) \]

The fields in the structure array \( S \) are the same as those used with \texttt{subsref}.

The \texttt{subsasgn} function lets you distinguish between assignments like
\[ A(i) = [] \] and \[ A(i) = B \], where \( B \) is empty. If the right-hand side of the assignment is the literal empty matrix [ ] and not a variable, the third input argument to \texttt{subsasgn} receives the string ' [ ] '. Therefore, you can distinguish between the two cases in your own \texttt{subsasgn} method.
Inheritance

As described earlier, MATLAB objects are always structures. This implementation facilitates inheritance, the process by which objects can acquire the behavior of other classes of objects. When one object (the child) inherits from another (the parent), the child object includes all the fields of the parent object and can call the parent’s methods.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing child objects to take advantage of code that exists for parent objects. The parent methods can access those fields that a child object inherited from the parent class, but not fields new to the child class.

There are two kinds of inheritance:

- Simple inheritance, in which a child object inherits characteristics from one parent class.
- Multiple inheritance, in which a child 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 another class, and adds new attributes of its own, uses simple inheritance. Inheritance implies that objects belonging to the child class have the same fields as the parent class, and usually additional fields. Therefore, methods associated with the parent class can operate on objects belonging to the child class. The methods associated with the child class, however, cannot operate on objects belonging to the parent class. You cannot access the parent’s fields directly from the child class; you must use accessor methods defined for the parent.

The constructor function for a class that inherits the behavior of another has two special characteristics:

- It typically calls the constructor function for the parent class to create the “inherited” fields.
- The calling syntax for the class function is slightly different, reflecting both the new class and the parent class.
A good example of simple inheritance is provided by the objects used in the
Control System Toolbox to analyze linear, time-invariant systems (LTI). The
parent class is called lti. There are three child classes, or subclasses,
corresponding to three different representations of LTI systems.

\begin{itemize}
  \item \texttt{tf} Transfer function
  \item \texttt{zpk} Zero, pole, gain
  \item \texttt{ss} State space
\end{itemize}

An \texttt{lti} object carries information that is independent of the particular
representation, including whether the system is continuous or discrete, the
value of the discrete sample time, and the names of the inputs and outputs.
The child objects carry the model-specific data for their representations. A \texttt{tf}
object carries coefficient vectors specifying the numerator and denominator of
a rational transfer function. A \texttt{zpk} object carries vectors containing the zeros,
poles and gain of the system. An \texttt{ss} object carries four matrices giving the state
space description of the system.

The statement
\begin{verbatim}
  L = lti(1,1)
\end{verbatim}
is not intended to be used directly. It merely creates a skeleton LTI object with
a zero sample time and empty input/output names. The statement
\begin{verbatim}
  T = tf(1, [1 0 -2 -5])
\end{verbatim}
creates a transfer function object that represents the continuous-time rational
transfer function
\[ \frac{1}{s^3 - 2s - 5} \]

Closer examination of the object \texttt{T} shows that it has four fields:

\begin{itemize}
  \item \texttt{T.num} The numerator, 1
  \item \texttt{T.den} The denominator, [1 0 -2 -5]
  \item \texttt{T.variable} 's'
  \item \texttt{T.lti} The \texttt{lti} portion
The \textit{lti} field of a transfer function object is inherited from its parent LTI object. In this case, it is the same as the skeleton object \texttt{L}. Overloaded \texttt{set} and \texttt{get} functions in the Control System Toolbox can be used to flesh out the object properties.

The following diagram illustrates this inheritance relationship:

The statements

\begin{align*}
S &= \text{ss}(T) \\
Z &= \text{zpg}(T)
\end{align*}

convert the transfer function representation to the other representations, retaining the \texttt{lti} portion, which is the same in all three representations.

The inheritance mechanism is implemented in the constructor functions for the child classes. The \texttt{tf} constructor, for example, includes the statement

\begin{align*}
L &= \text{lti}(Ny, Nu, Ts)
\end{align*}
to create an `lti` object with the appropriate parameters. The `tf` object is then created with the statement

```matlab
sys = class(sys, 'tf', L)
```

This use of `class` with three arguments assigns the appropriate class tag to the object and also indicates that it inherits from the parent object.

Inheritance can span more than one generation. That is, a child object can contain fields from both grandparent and parent objects (or more generations, if desired). In this case, the parent object can call grandparent methods, and the child object can call both parent and grandparent methods.

**Multiple Inheritance**

In the multiple inheritance case, a class of objects inherits attributes from more than one parent class. The child object gets fields from all the parent classes, as well as fields of its own.

Multiple inheritance can encompass more than one generation. For example, each of the parent objects could have inherited fields from multiple “grandparent” objects, and so on. Multiple inheritance is implemented in the constructors by calling `class` with more than three arguments.

```matlab
obj = class(structure, 'class_name', parent1, parent2, ...)
```

You can append as many parent arguments as desired to the class input list.
The following diagram shows how a hypothetical eco object might inherit fields from two parents, weather and plant.

Object A, class 'weather'
---
weather.rain: 0.0 0.0 0.15 1.2 0.0
weather.temp: 71 68 70 72 68

Object B, class 'plant'
---
plant.rain: 21 21 22 23 22
plant.fern: 4 4 4 4 4
plant.oak: 0.0 0.0 0.15 1.2 0.0

Object C, class 'eco'
---
weather.rain: 0.0 0.0 0.15 1.2 0.0
weather.temp: 71 68 70 72 68
plant.rain: 21 21 22 23 22
plant.fern: 4 4 4 4 4
plant.oak: 0.0 0.0 0.15 1.2 0.0

Inherited fields from A
---
weather.rain: 0.0 0.0 0.15 1.2 0.0
weather.temp: 71 68 70 72 68

Inherited fields from B
---
plant.rain: 21 21 22 23 22
plant.fern: 4 4 4 4 4

Fields specific to child object class
---
weather.tempavg: 69.8000
plant.fernavg: 21.8000
plant.oakavg: 4

Multiple parent classes may have associated methods of the same name. In this case, MATLAB uses the method associated with the parent that appears first in the class function call in the constructor function.
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. A rational object may use two polynomials for the numerator and denominator, for example. Because you can access object fields only from within a method, you can call a method for the contained object only from within a method for the outer object.

How MATLAB Calls Methods
When working with objects and methods, MATLAB uses a special set of rules to ensure that it calls the desired function. If at least one argument is an object, MATLAB looks at the argument list from left to right to determine which has the greatest precedence. (For operators of equal precedence, the leftmost is used.) It then uses these rules:

1. If the name of the called function is that of a MATLAB built-in function, MATLAB first checks to see if an overloaded version of the function exists for the class, and then for the parent class. If neither of these is the case, an error results.

2. If the function name is the same as the name of a class directory, MATLAB first checks to see if there is a converter function and if so, calls it. Otherwise, MATLAB calls the constructor for the class type.

3. If both 1 and 2 are false,
   a. If there is a method of the appropriate type, call it.
   b. If there is a method corresponding to the parent of the type, call it.
   c. If there is a function with the specified name on the search path, call it.
   d. Generate an error.

Private Methods and Private Functions
Class directories can have private directories associated with them. Such private directories can contain both private methods, which work on objects that belong to the class, and private functions, which do not work on objects but perform general computations. You can set up a private directory under a class.
directory the same way you create any private directory. Simply create a
directory named private under the @class_name directory.

**Debugging Object Methods**
You can use the MATLAB debugging commands with object methods, just as
you use them with any M-file. Simply include the class directory name before
the method name. For example,

```
    dbstop class/method
```

Note that while using the debugger, the default form of the `which` command can
see private functions and methods under the class directory. (The form `which`
always sees them.)
File I/O

Opening and Closing Files .......................... 15-3
Using the File Identifier (fid) ....................... 15-4
Closing a File ......................................... 15-4
Temporary Files and Directories .................... 15-5

Binary Files ........................................... 15-7
Reading Binary Files ................................. 15-7
Writing Binary Files ................................ 15-8

Controlling Position in a File ...................... 15-9
Understanding File Position ......................... 15-10

Formatted Files ..................................... 15-12
Reading Strings Line-By-Line from Text Files .... 15-12
Reading Formatted Text ............................... 15-13
Writing Text Files .................................... 15-15
MATLAB’s file input and output (I/O) functions read and write arbitrary binary and formatted ASCII files. They allow you to read data collected in other formats and to write out data for other programs or devices.

The low-level file I/O functions live in a directory called iofun in the MATLAB toolbox.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Opening and Closing</td>
<td>fopen</td>
<td>Open file.</td>
</tr>
<tr>
<td></td>
<td>fclose</td>
<td>Close file.</td>
</tr>
<tr>
<td>Binary I/O</td>
<td>fread</td>
<td>Read binary data from file.</td>
</tr>
<tr>
<td></td>
<td>fwrite</td>
<td>Write binary data to file.</td>
</tr>
<tr>
<td>Formatted I/O</td>
<td>fscanf</td>
<td>Read formatted data from file.</td>
</tr>
<tr>
<td></td>
<td>fprintf</td>
<td>Write formatted data to file.</td>
</tr>
<tr>
<td></td>
<td>fgetl</td>
<td>Read line from file, discard newline character.</td>
</tr>
<tr>
<td></td>
<td>fgets</td>
<td>Read line from file, keep newline character.</td>
</tr>
<tr>
<td>String Conversion</td>
<td>sprintf</td>
<td>Write formatted data to string.</td>
</tr>
<tr>
<td></td>
<td>sscanf</td>
<td>Read string under format control.</td>
</tr>
<tr>
<td>File Positioning</td>
<td>ferror</td>
<td>Inquire file I/O error status.</td>
</tr>
<tr>
<td></td>
<td>feof</td>
<td>Test for end-of-file.</td>
</tr>
<tr>
<td></td>
<td>fseek</td>
<td>Set file position indicator.</td>
</tr>
<tr>
<td></td>
<td>ftell</td>
<td>Get file position indicator.</td>
</tr>
<tr>
<td></td>
<td>frewind</td>
<td>Rewind file.</td>
</tr>
<tr>
<td>Temporary Files</td>
<td>tempdir</td>
<td>Get temporary directory name.</td>
</tr>
<tr>
<td></td>
<td>tempname</td>
<td>Get temporary file name.</td>
</tr>
</tbody>
</table>
Opening and Closing Files

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>Open file.</td>
</tr>
<tr>
<td>fclose</td>
<td>Close file.</td>
</tr>
</tbody>
</table>

Before reading or writing an ASCII or binary file you must open it with the `fopen` command.

```matlab
fid = fopen('filename', 'permission')
```

The `permission` string specifies the kind of access 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 MS-Windows that distinguish between text and binary files may require additional characters in the permission string, such as `'rb'` to open a binary file for reading.

`fopen` returns a file identifier (`fid`), the value you use to access the open file.

This `fopen` statement opens the data file named `penny.dat` for reading:

```matlab
fid = fopen('penny.dat', 'r')
```

### MATLAB File I/O Functions and ANSI Standard C

Many of the MATLAB file I/O functions are based on the I/O functions of the ANSI Standard C Library. If you know C, therefore, you are probably familiar with these routines. However, not all MATLAB file I/O commands work the same way as their C language counterparts. Check MATLAB command syntax and functionality using the online help facility or the online MATLAB Function Reference.
Using the File Identifier (fid)

The file identifier that fopen returns (if successful) is a nonnegative integer. This integer acts as a handle to the file, and is an argument to MATLAB file I/O functions.

There are times when fopen might fail. For example, fopen fails if you try to open a file that does not exist. If fopen fails, it does the following:

- It assigns –1 to the file identifier.
- It 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 may also provide information about errors.

It's good practice to test the file identifier each time you open a file. For example, this code loops until the user enters the name of a readable file:

```matlab
fid=0;
while fid < 1
    filename=input('Open file: ', 's');
    [fid, message] = fopen(filename, 'r');
    if fid == -1
        disp(message)
    end
end
```

Now assume that nofile.mat does not exist but that goodfile.mat does exist. On one system, the results are:

```
Open file: nofile.mat
```

```
Open file: goodfile.mat
```

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:

```matlab
status = fclose(fid);
```
This line closes all open files:

```matlab
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.

### Temporary Files and Directories

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tempdir</code></td>
<td>Get temporary directory name.</td>
</tr>
<tr>
<td><code>tempname</code></td>
<td>Get temporary file name.</td>
</tr>
</tbody>
</table>

You can create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary may mean only that the file is not backed up.

A function named `tempdir` 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:

```matlab
fid = fopen(tempname, 'w');
```
\textbf{NOTE} The filename that \texttt{tempname} generates is not guaranteed to be unique; however, it is likely to be so.
Binary Files

This section explains how to read from or write to binary files.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fread</td>
<td>Read binary data from file.</td>
</tr>
<tr>
<td>fwrite</td>
<td>Write binary data to file.</td>
</tr>
</tbody>
</table>

**Reading Binary Files**

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 in 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`.

```matlab
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 ASCII characters, transposing the data so it displays horizontally:

```matlab
disp(char(A'))
```

The `char` function causes MATLAB to interpret the contents of `A` as ASCII 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`:

```matlab
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 \texttt{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)

\textbf{NOTE} The meaning of a given precision can vary across different hardware platforms. For example, a `'uchar'` is not always 8 bits. \texttt{fread} also provides a number of more specific precisions, such as `'int8'` and `'float32'`. If in doubt, use these precisions, which are not platform dependent. Look up \texttt{fread} in online help for a complete list of precisions.

For example, if \texttt{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 \texttt{A}:

\begin{verbatim}
A = fread(fid,10,'float');
\end{verbatim}

\textbf{Writing Binary Files}

The \texttt{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:

\begin{verbatim}
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
\end{verbatim}

In this case, \texttt{fwrite} sets the \texttt{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 table below summarizes MATLAB functions for controlling the file position indicator:

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>feof</code></td>
<td>Determine if file position indicator is at end-of-file.</td>
</tr>
<tr>
<td><code>fseek</code></td>
<td>Set file position indicator.</td>
</tr>
<tr>
<td><code>ftell</code></td>
<td>Get file position indicator.</td>
</tr>
<tr>
<td><code>frewind</code></td>
<td>Reset file position indicator to beginning of file.</td>
</tr>
</tbody>
</table>

The `fseek` and `ftell` functions let you 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 an origin from which to calculate the move, specified as a string:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cof'</td>
<td>Current position in file</td>
</tr>
<tr>
<td>'bof'</td>
<td>Beginning of file</td>
</tr>
<tr>
<td>'eof'</td>
<td>End of file</td>
</tr>
</tbody>
</table>
Understanding File Position
To see how \texttt{fseek} and \texttt{ftell} work, consider this short M-file:

\begin{verbatim}
A = 1:5;
 fid = fopen('five.bin','w');
 fwrite(fid, A,'short');
 status = fclose(fid);
\end{verbatim}

This code writes out the numbers 1 through 5 to a binary file named \texttt{five.bin}. The call to \texttt{fwrite} specifies that each numerical element be stored as a \texttt{short}. Consequently, each number uses two storage bytes.

Now reopen \texttt{five.bin} for reading:

\begin{verbatim}
 fid = fopen('five.bin','r');
\end{verbatim}

This call to \texttt{fseek} moves the file position indicator forward six bytes from the beginning of the file:

\begin{verbatim}
 status = fseek(fid, 6, 'bof');
\end{verbatim}

This call to \texttt{fread} reads whatever is at file positions 7 and 8 and stores it in variable \texttt{four}:

\begin{verbatim}
 four = fread(fid, 1, 'short');
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline
File Position & bof & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & eof \\
\hline
File Contents & 0 & 1 & 0 & 2 & 0 & 3 & 0 & 4 & 0 & 5 & & \\
\hline
File Position Indicator & \uparrow & & & & & & & & & & & \\
\hline
\end{tabular}
\caption{File Position Table}
\end{table}
The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`:

```c
position = ftell(fid)
```

```
position = 8
```

This call to `fseek` moves the file position indicator back four bytes:

```c
status = fseek(fid, -4, 'cof');
```

<table>
<thead>
<tr>
<th>File Position</th>
<th>bof</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Contents</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File Position Indicator</th>
<th></th>
</tr>
</thead>
</table>

Calling `fread` again reads in the next value (3).

```c
three = fread(fid, 1, 'short');
```
Formatted Files

This section explains how to read from and write to formatted ASCII text files.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetl</td>
<td>Read line from file, discard newline character.</td>
</tr>
<tr>
<td>fgets</td>
<td>Read line from file, keep newline character.</td>
</tr>
<tr>
<td>fscanf</td>
<td>Read formatted data from file.</td>
</tr>
<tr>
<td>fprintf</td>
<td>Write formatted data to file.</td>
</tr>
</tbody>
</table>

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.

```matlab
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0
    line = fgetl(fid);
    matches = findstr(line, literal);
    num = length(matches);
    if num > 0
        y = y + num;
        fprintf(1, '%d:%s
', num, line);
    end
end
fclose(fid);
```

Given the following input datafile called badpoem:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflies or fleas.
```

Calling the `litcount` function with the string 'an' produces the output:

```
litcount('badpoem', 'an')
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,
```

**Reading Formatted Text**

The `fscanf` function is like the `fscanf` function in standard C. Both functions operate in a similar manner, reading a line of 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:

- `%s` to match a string
- `%d` to match an integer in base 10 format
- `%g` to match a double-precision floating-point value

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`:

```matlab
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`:

```matlab
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix `A`:

```matlab
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.

```matlab
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```
Writing 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

- `%e` for exponential notation
- `%f` for fixed point notation
- `%g` to 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 `exptab1e.txt`:

```
fid = fopen('exptab1e.txt','w');
fprintf(fid,'Exponential Function

');
fprintf(fid,'%6.2f %12.8f
',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:

```matlab
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:

```matlab
root2 = sprintf('The square root of %f is %10.8e.
',2,sqrt(2));
```
Symbols
%10-8, 10-9
& 10-25
| 10-25
~ operator 10-25
~ 10-25, 10-26
~ operator 10-25
~ 10-23

A
abs 6-36
absolute accuracy (ODE) 8-21
accuracy of calculations 10-17
adding
    cells to cell arrays 13-20
    fields to structure arrays 13-8
addition
    of matrices 4-6
addition operator 10-22
adjacency matrix 9-15
    and graphing 9-15
    Bucky ball 9-16
    defined 9-15
    distance between nodes 9-21
    node 9-15
    numbering nodes 9-17
administration
    license 2-44
advanced indexing 10-39
airflow modeling 9-21
algorithms

ODE solvers
    Adams-Bashworth-Moulton PECE 8-10
    Bogacki-Shampine (2,3) 8-10
    Dormand-Prince (4,5) 8-10
    modified Rosenbrock formula 8-11
    numerical differentiation formulas 8-11
aliases
    for MATLAB 2-2
all 10-27
AND operator
    rules for evaluating 10-25
angle 6-36
ans 10-17
answer, assigned to ans 10-17
any 10-27
Application Program Interface 1-5
arguments
    checking number of 10-12
        example 10-12
    defined for function 10-7
    for ODE file 8-6
    order 10-13, 10-15
    passed by reference 10-11
    passed by value 10-11
    passing variable number 10-14
arithmetic expressions 10-22
arithmetic operators 10-22
    combining with others in expressions 10-28
preference 10-22
array elements 4-4
arrays
    cell array of strings 11-7
    character 10-19, 11-4
dimensions
    inverse permutation 12-14
Index

indexing
  advanced 10-39
  multidimensional 12-3
  numeric
    converting to cell array 13-30
    of strings 11-5
    storage 10-39
arrow keys
  for editing 2-4
ASCII characters
  converting 11-5
ASCII files 15-12
  loading into workspace 2-24
  reading
    delimiter-separated text 2-38
    formatted text 15-13
    reading line-by-line 15-12
  saving 2-23
  writing 15-15
assignment statements 4-4, 10-15, 10-16
  using to build structure arrays 13-3
A-stable differentiation formulas 8-32
automatic scalar expansion 10-23

B
  ballode 8-42
  bandwidth of sparse matrix
    reducing 9-27
  base (numeric), converting 11-13
  base date 10-51
  base number conversion 11-3
  bicubic interpolation 5-12, 5-13
  bilinear interpolation 5-12, 5-13
  binary
    from decimal conversion 11-13
    binary files 15-7
  controlling data type of values read 15-8
  controlling number of values read 15-7
  reading 15-7
  writing to 15-8
  b t f u n directory 10-25
  blanks
    finding in string arrays 11-11
    removing from strings 11-6
  bof 15-9
  boundary condition
    ODE 8-40
  breakpoints 3-3
    and keyboard mode 3-17
    clearing 3-8, 3-13, 3-18
    setting 3-6, 3-11, 3-16
  Brusselator system (ODE example) 8-36
  brussode 8-36
  Buckminster Fuller dome 9-16
  Bucky ball 9-16

C
  C functions
    used for memory management 2-42
  calling context 10-11
  calling MATLAB functions
    compilation for later use 10-10
    how MATLAB searches for functions 10-10
    ODE solvers 8-6
    storing as pseudocode 10-11
  case 10-31
  case conversion 11-3
  cat 9-24, 12-2, 12-7, 12-8, 12-19
  catch function for error handling 10-48
cell
  indexing 13-19
    to access a subset of cells 13-23
Index

cell arrays
   accessing a subset of cells 13-23
   accessing data 13-21
      using content indexing 13-22
   applying functions to 13-26
   cell indexing 13-19
   concatenating 13-21
   content indexing 13-19
   converting to numeric array 13-30
   creating 13-18
      using assignments 13-19
      with cell's function 13-21
      with curly braces 13-21
   defined 13-2
   deleting cells 13-23
   deleting dimensions 13-23
   displaying 13-20
   expanding 13-20
   flat 13-28
   indexing 13-19
   multidimensional 12-19
   nested 13-28
      building with the cell's function 13-29
      indexing 13-29
   of strings 11-7
      comparing strings 11-10
   of structures 13-31
   organizing data 13-27
   overview 13-18
   preallocation 10-61, 13-21
   reshaping 13-24
   to replace comma-separated list 13-24
   visualizing 13-20

   cell class 14-2
   cell data type 10-20
   cell disp 13-20
   cellplot 13-20

   cells 13-18, 13-21, 13-28
   char 14-6
   char class 14-2
   char data type 10-20, 15-8
   character arrays 10-19, 11-4
      two-dimensional 11-5
   characteristic polynomial 5-3
   characteristic roots of matrix 5-3
   chol 9-24, 9-30
   Cholesky factorization 4-26
      for sparse matrices 9-30
   class 14-6
   class directories 14-3
   classes 10-19, 14-2
      cell 14-2
      char 14-2
      constructors 14-3, 14-4
      double 14-2
      methods (object-oriented) 14-2
      overview 14-2
      sparse 14-2
      struct 14-2
      toolbox 14-2
   clear 10-11, 10-62
   clearing M-files 3-3
   closest point searches 5-23
   closing files 15-2, 15-3
   cof 15-9
   colmd 9-27, 9-28
   colon operator 4-8, 10-22, 12-10
      for multidimensional array subscripting 12-10
      in subscripts 8-28
      to access subsets of cells 13-23
      used to index a page 12-16
      used with scalar expansion 12-6
   colperm 9-26
   col per m 9-26
   column vector 4-4
for polynomial roots representation 5-3
indexing as 10-40
of event locations (ODE) 8-31
columns 4-29
deleting 10-39
comma
to separate function arguments 10-7
command line
editing 2-3
Command Window 2-3
comma-separated list
using cell arrays 13-24
comments 10-9
comparing
interpolation methods 5-13
sparse and full matrix storage 9-11
strings 11-9
comparing strings 11-9
complex conjugate transpose 4-8
complex conjugate transpose operator 10-22
complex values
in sparse matrix 9-5
computational functions
applying to cell arrays 13-26
applying to multidimensional arrays 12-15
applying to sparse matrices 9-23
applying to structure fields 13-8
in M-file 10-3
computer 10-18
computer type 10-18
concatenating
cell arrays 13-21
strings 10-43
concatenation 10-38
condense 9-3
condition
dimension compatability 4-14
conditional statements 10-12
conditions
for ODEs
boundary 8-40
initial 8-3, 8-5
confidence intervals 6-27
constructors 14-3, 14-4
content indexing 13-19
to access cell contents 13-22
contents
of sparse matrix 9-11
Contents file 10-8
contiguous memory 2-42
Continue
in debugger 3-5
continuing statements on multiple lines 2-6
continuous extension (ODE solvers) 8-12
contour 5-22
contour plots
to compare interpolation methods 5-14
control keys
for editing 2-4
conv 5-4, 13-25
converter functions 14-6
converting
and handling dates 10-51
ASCII characters 11-5
base numbers 11-3
cases of strings 11-3
numbers 11-13
strings 11-3, 11-13, 15-2
convex hull 5-24
convhull 5-24
convolution 5-4
corrcoef 6-10
correlation coefficients 6-9
cos 10-5
count variable 15-16
cov 6-9
covariance 6-9
creating
cell array 13-18
multidimensional array 12-4
ODE files 8-14
sparse matrix 9-7
string 11-4
string array 11-5
structure array 13-3
cross 12-15
cubic interpolation
  multidimensional 5-16
  one-dimensional 5-10
cubic spline interpolation 5-10
curly braces
  for cell array indexing 13-19, 13-21
  to build cell arrays 13-21
  to nest cell arrays 13-28
curve fitting 5-6
  confidence intervals 6-27
Cuthill-McKee, reverse ordering 9-27

data files
  exchanging between platforms 2-39
data fitting 6-20
  confidence intervals 6-27
  error bounds 6-27
  exponential fit 6-24
  exponential fits 6-24
  polynomial fits 6-20
data gridding
  multidimensional 5-17
data normalization 6-21
data organization
  cell arrays 13-27
  multidimensional arrays 12-17
  structure arrays 13-10
data types 10-19
  cell 10-20
  char 10-20
  double 10-20
  for image processing 10-19
  for toolbox creation 10-19
  precision
    char 15-8
    float 15-8
    long 15-8
    short 15-8
    uchar 15-8
  precisions 15-8
  sparse 10-20
  sparse matrices 10-19
  struct 10-20
  uint 10-20
  user defined 14-2
  UserObject 10-20
data analysis
  finite differences 6-11
  triangulation 5-19
data
  binary 15-2
  filter 6-29
  Harwell-Boeing format 9-10
  monotonic 5-12
  multivariate 6-3
  pre-processing 6-12
  sorting 6-7
date
  base 10-51
  conversions 10-52
Index

formats 10-51
number 10-51
string
  vector of input 10-54
date 10-56
datenum 10-52, 10-53
dates
  handling and converting 10-51
datet 10-52, 10-54
date 10-52
dbdon 3-17
dbusop 3-16
dbup 3-17
deblang 11-6
debugger
  changing workspace context 3-8, 3-12
  command line 3-15
  command summary 3-15
  Continue 3-5
  ending a session 3-10, 3-14
  keyboard mode 3-16
  overview 3-2
  pause 3-7, 3-12
  quitting 3-20
  stack 3-2, 3-8
  step 3-5
  stepping through code 3-16
  techniques 3-2
  yellow arrow 3-7
decimal representation
  to binary 11-13
  to hexadecimal 11-13
decomposition
  eigenvalue 4-36
  Schur 4-39
  singular value 4-40
deconv 5-4
deconvolution 5-4
Delaunay triangulation 5-20
  closest point searches 5-23
deleting
  cells from cell array 13-23
  fields from structure arrays 13-8
deletion operator 10-39
delimiter
  in string 11-12
delimiter-separated text files 2-38
demos
  MATLAB 1-6
density of sparse matrix 9-6
derivative
  of polynomial 5-5
descriptive statistics 6-7
det 4-22
determinant 4-22
diag 9-24
diagonal
  creating sparse matrix from 9-8
  of a matrix 4-11
diary 2-37, 2-40
diff 6-11
difference
  between successive vector elements 6-11
difference equations 6-29
differential equations. See ODE solvers
dimension for cat 12-7
dimension compatibility 4-14
dimensions
  deleting 13-23
  permuting 12-13
  removing singleton 12-12
direct methods for systems of equations 9-33
directories
Index

bitfun 10-25

class 14-3

Contents.m file 10-8

help for 10-8

MATLAB 1-5

private 10-36

discrete Fourier transform 6-31

disp 13-7, 13-15

dispatching priority 10-10

methods (object-oriented) 14-27

display 14-8

displayed output 14-8

displaying

   cell arrays 13-20

   error and warning messages 10-49

   field names for structure array 13-4

   sparse matrices 9-13

distance between nodes 9-21

division

   matrix 4-14

   of polynomials 5-4

doc 2-34

documentation

   how to use 1-6

dot product 4-8

double

   data type 10-20, 15-8

double class 14-2, 14-6

double precision 15-8

double-precision matrix 10-19

dsearch 5-23

E

editor 2-35

   accessing 10-3

   arrow and control keys 2-4

command line 2-3

   for ASCII files 2-36

   for creating M-file 10-2, 10-3

MATLAB 3-5

   using to create M-files 2-36

eig 4-36, 5-3, 12-16

eigenvalue

   decomposition 4-36

   eigenvalues 4-36

   of sparse matrix 9-36

eigenvector 4-36

element-by-element organization for structures 13-13

else 10-29, 10-30

elseif 10-29, 10-30

empty arrays

   and if statement 10-30

   and relational operators 10-24

   and while loops 10-33

empty matrices 10-46

end of file 15-9

eof 15-9

eps 10-17

epsilon 10-17

equal to operator 10-23

error

   bound

      for data fit 6-27

      handling 10-48

      tolerance (ODE ) 8-17, 8-21

error 10-6

errors 10-48

   displaying 10-49

I/O 15-2

   memory 2-42

eval

   for error handling 10-48
Index

evaluating
  polynomials in matrix sense 5-4
  string containing function name 10-43
  string containing MATLAB code 11-14
  string containing MATLAB expression 10-43

event location (ODE) 8-17, 8-30
examples
  adjacency matrix (sparse) 9-15
  airflow modeling 9-21
  brussode 8-36
  Bucky ball 9-16
  checking number of function arguments 10-12
  Delaunay triangulation 5-20
  f emode 8-40
  for 10-34
  function 10-6
  if 10-29
  interpolation 5-13
  M-file for structure array 13-10
  ODE solvers 8-33
  orbitode 8-43
  profiling 3-24
  rigidode 8-33
  script 10-5
  second difference operator 9-7
  sparse matrix 9-7, 9-15
  switch 10-32
  theoretical graph (sparse) 9-15
  van der Pol 8-6
    extra parameters 8-16
    stiff 8-8
  vdpode 8-34
  vectorization 10-59
  while 10-33
exclusive OR operator 10-26

execution
  pausing 10-57

script 10-5
expanding
  cell arrays 13-20
  structure arrays 13-4
  exponential fit to data 6-24
  exponentials
    matrix 4-33
  expressions
    and scalar expansion 10-23
    arithmetic 10-22
    involving empty arrays 10-24
    logical 10-25
    most recent answer 10-17
    overloading 14-10
    relational 10-23
  external program
    running from MATLAB 10-58

eye 4-11, 9-23

F
factorization 9-28
  Cholesky 4-26
  for sparse matrices 9-28
    Cholesky 9-30
    LU 9-28
    triangular 9-28
  incomplete 9-32
  LU 4-27
  positive definite 4-26
  QR 4-30
fast Fourier transform See Fourier transform, fast
fclose 15-4
f emode 8-40
f emode example 8-40
f error 15-2
fft 6-37, 6-38
FFTs. See Fourier transform, fast
fields 13-3
accessing with getfield 13-7
accessing data within 13-6
accessing multiple fields 13-7
adding to structure array 13-8
applying functions to 13-8
all like-named fields 13-9
assigning data to 13-3
assigning with setfield 13-7
defined 13-3
deleting from structures 13-8
indexing within 13-6
initializing 13-5
names 13-4
size 13-8
writing M-files for 13-9
file identifier 15-3, 15-4, 15-5
filenames
  contained in variables 2-24
  wildcards for loading 2-24
files
  ASCII 15-12
    loading into workspace 2-24
    reading delimiter-separated text 2-38
    reading formatted text 15-13
    writing 15-15
  beginning of identifier 15-9
  binary 15-7
    controlling data type values read 15-8
    controlling number of values read 15-7
    reading 15-7
    writing to 15-8
  bof 15-9
  closing 15-2, 15-3, 15-4
  cof 15-9
  Contents.m 10-8
  controlling position 15-9
  current position 15-9
  end of identifier 15-9
  exchanging between platforms 2-39
  formatted 15-12
  I/O 15-2
  identifiers 15-4
  license.dat 2-45
  local options 2-45
  matlabrc.m 2-41
  opening 15-2, 15-3
    permissions 15-3
  positioning 15-2
  saving ASCII 2-23
  start-up 2-41
  startup.m 2-41
  temporary 15-2, 15-5
  text 15-12
  fill-in of sparse matrix 9-20
  filtering 6-29
  find function
    and sparse matrices 9-14
    and subscripting 10-27
  finding
    nonzero elements 6-13
    substring within a string 11-12
  finite differences 6-11
  finite element discretization (ODE example) 8-40
  first-order differential equations
    representation for ODE solvers 8-5
  flag argument (ODE) 8-18
  flat structure arrays 13-16
  FLEXlm license manager 2-44
  float 15-8
  floating-point number
largest 10-17
smallest 10-17
floating-point operations, count 10-18
floating-point relative accuracy 10-17
f l o p s 10-18
flow control 10-29
   and relational operators 10-24
   e l s e 10-29
   e l s e i f 10-29
   f o r 10-33
   i f 10-29
   s w i t c h 10-31
   w h i l e 10-33
f m i n 7-7, 7-9
f m i n s 7-8
f o p e n 15-3
f o r 10-33, 13-30
   example 10-34
   l o o p
      indexing 10-34
      matrices as indices 10-34
      nested 10-34
      syntax 10-33
      vectorization 10-29
format
   d a t e 10-51
   of output display 2-5
f o r m a t 2-5
formatted files 15-12
Fourier analysis 6-31
Fourier transform
   f a s t
      F F T - b a s e d interpolation 5-11
      specifying length 6-38
f p l o t 7-4
fragmentation, reducing 10-61
f r e a d 15-7
f s c a n f 15-13
f s e e k 15-9
f t e l l 15-9
f u l l 9-24, 9-27
function 10-5
   applying
      to multidimensional structure arrays 12-21
   applying to structure contents 13-8
   body 10-3, 10-8
   calling context 10-11
   characteristics 10-2
   clearing from memory 10-11
   contents 10-6
   defined 10-2
   definition line
      for subfunction 10-35
   dispatching priority 10-10
   example 10-6
   executing function name string 10-43
   i n t e r p f t 5-11
   l o g i c a l 10-26
   minimizing 7-7
   multiple ones with same name 2-26
   name 10-9
   name resolution 10-10
   optimization 10-59
   order of arguments 10-13, 10-15
   passing variable number of arguments 10-14
   primary 10-35
   private 10-36
   shadowed 2-26
   storing as pseudocode 10-11
   to create arrays 12-6
   what happens when called 10-10
   workspace 10-11
function definition line 10-7
   defined 10-3
function functions 7-1
functions
  %10-8, 10-9
  ~10-26
  ~=10-23
  abs 6-36
  all 10-27
  angle 6-36
  ans 10-17
  any 10-27
applying to cell arrays 13-26
ballode 8-42
brussode 8-36
cat 9-24, 12-2, 12-7, 12-8, 12-19
cells 13-18, 13-21, 13-28
char 14-6
c chol 9-24, 9-30
class 14-6
clear 10-11, 10-62
colmd 9-27, 9-28
colperm 9-26
computer 10-18
cond est 9-3
contour 5-22
conv 5-4, 13-25
converter 14-6
converters 14-6
convhull 5-24
cor rcoef 6-10
cos 10-5
cov 6-9
cross 12-15
date 10-56
dat enum 10-52, 10-53
date str 10-52, 10-54
dat vec 10-52
dbd down 3-17
dbqui t 3-20
dbst op 3-16
dbup 3-17
deb l ank 11-6
deconv 5-4
det 4-22
di ag 9-24
di ary 2-37, 2-40
diff 6-11
disp 13-7, 13-15
disp lay 14-8
docr 2-34
double 14-6
dsearch 5-23
eig 4-36, 5-3, 12-16
eps 10-17
er rror 10-6
eye 4-11, 9-23
fclose 15-4
f emode 8-40
ferror 15-2
fft 6-37, 6-38
fi el dnames 13-4
fi el ds 13-4
find 9-14
flops 10-18
fmi n 7-7, 7-9
fmi ns 7-8
fopen 15-3
for 13-30
for m at 2-5
fpl ot 7-4
fread 15-7
fscanf 15-13
fseek 15-9
Index

ftell 15-9
full 9-24, 9-27
fzero 7-10
getfield 13-7
gplot 9-15
griddata 5-22
hcureve 7-15
hel p 2-31, 10-8
humps 7-3
i 10-17
if 10-24
lnf 10-18
inferior to 14-16
interp1 5-9
interp2 5-12
interpn 5-15
interpn 5-16
inv 4-22
ipermute 12-2, 12-14
isa 14-6
isempty 10-24
isfinite 14-16
isinteger 12-9
isletter 11-11
isnan 6-13, 10-27
isspace 11-11
j 10-17
kron 4-11
load 9-10, 10-62
log10 6-25
lookfor 2-32, 10-3, 10-8
lu 9-28, 9-29
mat2str 11-14
max 9-24
mean 12-15
meshgrid 5-12, 5-15, 5-22
min 6-9
NaN 10-18
nargin 10-12
nargout 10-12
ndgrid 5-16, 5-17, 12-2
ndims 12-2, 12-9
nested 10-63
nnz 9-11, 9-13
nonzeros 9-11
norm 9-11, 9-13
ode113 8-10
ode115s 8-8, 8-11, 8-17, 8-29, 8-32, 8-40, 8-51
ode23 8-10, 8-51
ode23s 8-11, 8-29, 8-30, 8-41, 8-49, 8-50, 8-51
ode45 8-6, 8-8, 8-10, 8-23, 8-24, 8-49, 8-51
odeget 8-21
odephas2 8-24
odephas3 8-24
odeplot 8-24
odeset 8-20
ones 9-23, 12-6
optimization 7-7
or 10-24
orbitsode 8-43
overloading 14-10, 14-13
pack 2-42, 10-62
pcode 10-11
permut e 12-2, 12-13
pi 10-17
pi* 4-23
pol ar 10-5
poly 5-3
polyder 5-5
polyfit 5-6, 6-20, 6-23, 6-25, 6-27
polyval 5-4, 6-23, 6-25, 6-27
profile 3-22
qr 4-30, 9-31
quad 7-14, 8-47
quad8 7-14, 8-47
quit 10-62
rand 9-23
randn 12-6
realmax 10-17
realmin 10-17
repmat 12-6
reshape 12-11, 13-24
residue 5-7
rigidode 8-33
rmfield 13-8
roots 5-3
round 5-10
save 9-10, 10-62
schur 4-39
setfield 13-7
shiftdim 12-2
sin 10-5, 12-15
size 9-23, 12-9, 13-8
sort 9-27
sparse 9-6, 9-24
spconvert 9-10
spdiags 9-8
spieye 9-24, 9-26, 9-29
spones 9-27
spparms 9-27, 9-34
sprand 9-24
sprintf 15-16
spy 9-13
sqrtm 4-33
squeeze 12-2, 12-12, 12-16
sscanf 15-14
strcmp 13-3, 13-5, 13-15
subassign 14-5
subref 14-20
sum9-24, 9-27, 12-15
superiorto 14-16
svd 4-41
svds 9-37
switch 10-24
symmd 9-27
symmmd 9-29
symcm9-27, 9-28, 9-29
tempdir 15-5
tempname 15-5
transpose 12-14
tsearch 5-23
unwrap 6-37
varargin 10-14, 13-26
varargout 10-15
vdpode 8-34
version 10-18
voronoi 5-23
while 10-24
whos 9-11, 10-62, 12-9
xor 10-26
zeros 9-23, 12-6
functions, computational
applying to structure fields 13-8
fzero 7-10

G
Gaussian elimination 4-26, 4-27
general questions, ODE solvers 8-47
geodesic dome 9-16
getfield 13-7
global variables 10-16
rules for use 10-17
INDEX

I-14

gplot 9-15
graph
   characteristics 9-20
   defined 9-15
   of profiler results 3-25
   theoretical 9-15
greater than operator 10-23
greater than or equal to operator 10-23
griddata 5-22
H
H1 line 10-7
   and help p command 10-3
   and lookfor command 10-3
   defined 10-3
Handle Graphics 1-4
handling and converting dates 10-51
Harwell-Boeing data format 9-10
hcurve 7-15
help 2-31, 10-8
   and H1 line 10-3
help text
   creating for directories 10-8
   defined 10-3
hexadecimal
   converting from decimal 11-13
humps 7-3

I
I/O
   binary 15-2
   error status 15-2
   formatted 15-2
   functions 15-2
   identity matrix 4-11
if 10-24, 10-29
   and empty arrays 10-30
   example 10-29
   nested 10-30
image processing
   data type 10-19
   images 2-42
   imaginary unit 10-17
importing
   sparse matrix 9-10
improving solver performance 8-17
incomplete factorization 9-32
indexing 10-37
   advanced 10-39
   cell array 13-19
   content 13-19
   for loops 10-34
   multidimensional arrays 12-10
   nested cell arrays 13-29
   nested structure arrays 13-16
   structures within cell arrays 13-31
   within structure fields 13-6
indices
   how MATLAB calculates 10-42
inf 10-18
inferior to 14-16
infinity (represented in MATLAB) 10-18
inheritance 14-22
   multiple 14-25
   simple 14-22
initial condition 8-14
   defined 8-5
   example 8-6
   initial condition vector 8-6
   ODE 8-3
initial value problem 8-2
   defined 8-5
initial values
  defining in ODE file 8-14
  returned by 8-15
inner product 4-7
input
  from keyboard 10-57
  obtaining from M-file 10-57
input arguments
  defined by function 10-7
integer data type 15-14
integers
  changing to strings 11-13
integration
  double 7-15
  numerical 7-14
  See also ordinary differential equation solvers
interactive user input 10-57
interpolation
  comparing methods 5-13
  cubic 5-10
  cubic spline 5-10
  defined 5-9
  FFT-based 5-11
  memory 5-11
  multidimensional 5-15, 5-16
    cubic 5-16
    linear 5-16
    nearest neighbor 5-16
  one-dimensional 5-9
    cubic 5-10
    cubic spline 5-10
    linear 5-10
    nearest neighbor 5-10
  polynomial 5-9
  smoothness 5-11
  speed 5-11
  three-dimensional
    nearest neighbor 5-15
    tricubic 5-15
    trilinear 5-15
  two-dimensional
    bicubic 5-12, 5-13
    bilinear 5-12, 5-13
    nearest neighbor 5-12, 5-13
interpreter
  MATLAB 2-3
  inv 4-22
  inverse 4-22
  inverse permutation of array dimensions 12-14
  isequal 12-2, 12-14
  isalpha 14-6
  isempty 10-24
  isnan 10-27
  isnan 10-27
  iterative methods
    for sparse matrices 9-35
    for systems of equations 9-33
J
  Jacobian matrix (ODE) 8-17, 8-25
    constant Jacobian 8-26
    evaluated analytically 8-27
    sparsity pattern 8-27
    vectorized computation 8-27
joining 10-38
Jordan Canonical Form 4-38

K
keyboard mode 3-16
keyboard statements 3-2
keys
   arrow 2-4
   control 2-4
kinds of M-files 10-2
kron 4-11
Kronecker tensor product 4-11

L
lasterr
   and error handling 10-48
least squares 9-31
less than operator 10-23
less than or equal to operator 10-23
letter
   corresponding ASCII value 11-5
   finding in string 11-11
library
   mathematical function 1-5
license manager 2-44
license.dat 2-45
linear algebra
   and matrices 4-2
linear interpolation 5-10
   multidimensional 5-16
linear systems of equations
   direct methods 9-33
   iterative methods 9-33
   sparse 9-33
   linear transformation 4-4
   linear-in-the-parameters regression 6-17
LM_LICENSE_FILE 2-45
load 9-10, 10-62
loading
   ASCII files 2-24
   MAT-files 2-23
   using wildcards 2-24
local variables 10-16
log 6-25
logarithm analysis with a second-order model 6-26
logical expressions 10-25
   and subscripting 10-27
logical functions 10-26
logical operators 10-25
   combining with others in expressions 10-28
   rules for evaluation 10-25
long 15-8
long integer 15-8
lookfor 2-32, 10-3, 10-8
   and H1 line 10-3
loops
   for 10-33
   while 10-33
lu 9-28, 9-29
LU factorization 4-27
   for sparse matrices and reordering 9-28

M
magnitude 6-36
Maple 4-38
mass matrix (ODE) 8-17, 8-29
   constant mass matrix 8-30
   returned by ODE file 8-30
MAT-files 2-40
   loading 2-23
   mathematical functions
finding zeros 7-7
library 1-5
minimizing 7-7, 7-9
numerical integration 7-14
of one variable 7-7
    finding zeros 7-10
of several variables 7-8
plotting 7-4
quadrature 7-14
representing in MATLAB 7-3
mathematical operations
    on sparse matrices 9-23
MathWorks, The
    home page 1-8
MATLAB
    alias 2-2
    Application Program Interface 1-5
    classes 14-2
    Command Window 2-3
demos 1-6
    editor 3-5
    Handle Graphics 1-4
    history 1-3
    interpreter 2-3
    language 1-4
    mathematical function library 1-5
    overview 1-3
    path 2-26, 2-41
    product family 1-9
    programming
        debugging 3-1
        functions 10-5
        M-files 10-2
        quick start 10-2
        scripts 10-5
    representing functions 7-3
    search path 2-26
    shortcuts 2-2
    starting 2-2
    start-up files 2-41
    structures 14-4
    version 10-18
    working environment 1-4
matl ab c. m file 2-41
matrices 10-38
    addition 4-6
    and linear algebra 4-2
    concatenation 10-38
    deleting rows and columns 10-39
diagonal of 4-11
    dimension compatibility 4-14
division 4-14
    empty 10-46
    full to sparse conversion 9-2, 9-6
    identity 4-11
    multiplication 4-8
    orthogonal 4-29
    sparse 10-19
    subtraction 4-6
    symmetric 4-7
    triangular 4-26
matrix
    as index for for loops 10-34
    characteristic roots 5-3
double-precision 10-19
elements 4-4
    iterative methods 9-35
See also matrices
    matrix exponentials 4-33
    matrix multiplication 4-8
    matrix power operator 10-22
    matrix powers 4-33
    max 9-24
    mean 12-15
memory 2-42
  and function workspace 10-11
management 2-42, 10-62
Out of Memory message 10-66
reducing fragmentation 10-61
usage returned by ps 2-42
use of variables 10-63
meshgrid 5-12, 5-15, 5-22
methods (object-oriented) 14-2, 14-19
  how called 14-27
  indexing objects within 14-19
M-file
  comments 10-9
  contents 10-3
  creating
    quick start 10-2
  creating with an editor 2-36
  creating with text editor 10-3
  dispatching priority 10-10
  for ODE solvers (ODE file) 8-6
  impact of clearing on breakpoints 3-3
  kinds 10-2
  obtaining input interactively 10-57
  obtaining keyboard input 10-57
  optimization 10-59
  overview 10-3
  pausing during execution 10-57
primary function 10-35
profiling 3-21
subfunction 10-35
superseding existing names 10-36
  to operate on structures 13-9
  to represent mathematical functions 7-3
M-files 2-39
Microsoft Windows
  MATLAB use of system resources 10-64
min 6-9
minimal norm 4-24
minimizing functions
  of one variable 7-7
  of several variables 7-8
  setting minimization options 7-9
minimum degree ordering 9-27
missing values 6-12
monotonic data
  for interpolation 5-12
Moore-Penrose pseudoinverse 4-23
multidimensional arrays
  applying functions 12-15
    element-by-element functions 12-15
    matrix functions 12-16
    vector functions 12-15
  cell arrays 12-19
  computations on 12-15
  creating 12-4
    at the command line 12-5
    with functions 12-6
    with the cat function 12-7
  defined 12-2
  extending 12-5
  format 12-9
  indexing 12-10
    avoiding ambiguity 12-10
    with the colon operator 12-10
  interpolation 5-15, 5-16
  number of dimensions 12-9
  organizing data 12-17
  permuting dimensions 12-13
  removing singleton dimensions 12-12
  reshaping 12-11
  size of 12-9
  storage 12-9
  structure arrays 12-20
    applying functions 12-21
subscripts 12-3
multidimensional data gridding 5-17
multidimensional interpolation 5-15, 5-16
cubic 5-16
linear 5-16
nearest neighbor 5-16
multiple conditions for switch 10-32
multiple inheritance 14-25
multiple regression 6-19
multiplication
matrix 4-8
of polynomials 5-4
multiplication operator 10-22
multistep solver (ODE) 8-10
multivariate data 6-3

N
names
for functions 10-9
for variables 10-16
functions
resolving 10-10
structure fields 13-4
superseding 10-36
NaN 6-12, 10-18
nargin 10-12
nargout 10-12
ndgrid 5-16, 5-17, 12-2
ndims 12-2, 12-9
nearest neighbor interpolation 5-10, 5-12, 5-13, 5-15
multidimensional 5-16
nesting
cell arrays 13-28
for loops 10-34
functions 10-63
if statements 10-30
structures 13-15
newlines in string arrays 11-11
nnz 9-11, 9-13
nodes 9-15
distance between 9-21
numbering 9-17
nonzero elements
number of 9-11
nonzero elements of sparse matrix 9-11
maximum number in sparse matrix 9-7
storage 9-5, 9-11
values 9-11
visualizing with spy plot 9-19
nonzeros 9-11
norm
minimal 4-24
nor m4 12
normalizing data 6-21
not equal to operator 10-23
Not operator
rules for evaluating 10-26
Not-a-Number 10-18
now 10-56
null 4-21
number of arguments 10-12
numbers
changing to strings 11-13
date 10-51
time 10-51
numerical integration 7-14
nzmax 9-11, 9-13

O
object-oriented programming 14-2
converter functions 14-6
inheritance 14-22
inheritance:multiple 14-25
inheritance:simple 14-22
method dispatching priority 14-27
object precedence 14-16
overloading 14-10, 14-13
subscripting 14-20
overview 14-2
private methods 14-27
See also classes and objects
objects 14-2
and arrays 14-16
indexing 14-19
inheritance 14-22
overview 14-2
precedence 14-16
ODE solver properties
error tolerance 8-17, 8-21
absolute accuracy 8-21
AbsTol 8-22
relative accuracy 8-21
RelTol 8-22
event location 8-17, 8-30
Events 8-30, 8-31
Jacobian matrix 8-17, 8-25
Jacobian 8-26
JConstant 8-26
JPattern 8-26
Vectorized 8-26, 8-27
mass matrix 8-17, 8-29
Mass 8-30
MassConstant 8-30
modifying property structure 8-20
ode15s 8-17
BDF 8-32
MaxOrder 8-32
odeset function 8-20
querying property structure 8-21
questions and answers 8-48
smoothing output 8-24
solution components for output function 8-24
solver output 8-17, 8-23
OutputFcn 8-23
OutputSel 8-23, 8-24
Refine 8-23, 8-24
Stats 8-23, 8-25
specifying (overview) 8-12
step size 8-17, 8-28
InitialStep 8-28, 8-29
MaxStep 8-28
ODE solvers
basic example
nonstiff problem 8-7
stiff problem 8-8
calling 8-6
different kinds of systems 7-12, 8-50
examples 8-33
flag argument 8-18
general questions 8-47
multistep solver 8-10
nonstiff solvers 8-10
obtaining performance statistics 8-13
obtaining solutions at specific times 8-12
one-step solver 8-10
overview 8-10
properties 8-48
questions and answers
different kinds of systems 8-50
properties 8-48
time steps 8-47
quick start 8-3
representing problems 8-5
rewriting problem as first-order system 8-6
See Also ODE files, ODE solver properties
solution array 8-12
stability 8-32
stiff problems 8-8
stiff solvers 8-10
syntax, basic 8-11
time
    span vector 8-11
time interval 8-6
time steps 8-47
troubleshooting 8-49
van der Pol example
    extra parameters 8-16
    nonstiff 8-6
    stiff 8-8
ode113 8-10
    description 8-10
ode15s 8-8, 8-11, 8-29, 8-32, 8-40, 8-51
    description 8-11
    properties 8-17
ode23 8-10, 8-51
    description 8-10
ode23s 8-11, 8-29, 8-30, 8-41, 8-49, 8-50, 8-51
    description 8-11
ode45 8-6, 8-8, 8-10, 8-23, 8-24, 8-49, 8-51
    description 8-10
odeget 8-21
odephas2 8-24
odephas3 8-24
odeplot 8-24
odeprint 8-24
odeset 8-20
offsets
    for indexing 10-42
one-dimensional interpolation 5-9, 5-10
    cubic spline 5-10
    linear 5-10
    nearest neighbor 5-10
ones 9-23, 12-6
one-step solver (ODE) 8-10
online help 2-32, 2-34
online help. See help, help text
opening files 15-2, 15-3
operator 10-22
    & 10-25
    | 10-25
    ~ 10-25
    addition 10-22
    applying to cell arrays 13-26
    applying to structure fields 13-8
    arithmetic 10-22
    colon 8-28, 10-22, 12-6, 12-10, 12-16, 13-23
    combining 10-28
    complex conjugate 10-22
    deletion 10-39
    equal to 10-23
    greater than 10-23
    greater than or equal to 10-23
    less than 10-23
    less than or equal to 10-23
    logical 10-25
    matrix power 10-22
    multiplication 10-22
    not equal to 10-23
    power 10-22
    precedence 10-22
    relational 10-23
    second difference 9-7
    subtraction 10-22
    transpose 10-22
    unary minus 10-22
    unary plus 10-22
operators
    | 10-25
    ~ 10-25
colon 4-8
overloading 14-3, 14-10, 14-11
semicolon 2-6
optimization 10-59
practicalities 7-12
preallocation
array 10-61
troubleshooting 7-12
vectorization 10-59
options
minimization 7-9
or 10-24
OR operator
rules for evaluating 10-25
or bitode 8-43
order of function arguments 10-13, 10-15
Ordinary Differential Equation
coding as M-file 8-6
coding to return initial values 8-15
creating 8-14
defining initial values 8-14
guidelines for creating 8-16
output 8-15
overview 8-14
passing additional parameters 8-16
template 8-18
ordinary differential equation solvers. See ODE solvers
ordinary differential equations
defined 8-5
rewriting for ODE solvers 8-5
organizing data
cell arrays 13-27
multidimensional arrays 12-17
structure arrays 13-10
origin
fseek argument 15-9
orthogonal matrix 4-29
orthogonalization 4-26
orthonormal columns 4-29
otherwise 10-31
Out of Memory message 10-66
outer product 4-7
output
controlling format 2-5
displayed 14-8
suppressing 2-6
output arguments
defined by function 10-7
output properties, ODE solvers 8-22
overdetermined systems of simultaneous linear equations 4-17
overloading 14-20
functions 14-10, 14-13
operators 14-3, 14-11
overriding operator precedence 10-22

P
pack 2-42, 10-62
page subscripts 12-3
parentheses
for input arguments 10-7
to override operator precedence 10-22
par et o 3-25
Pareto chart
of profiler results 3-25
partial fraction expansion 5-7
partial pivoting 4-28
parts of a function 10-6
passing arguments
by reference 10-11
by value 10-11
path
  MATLAB 2-26, 2-41
  search 2-26
pause
  in debugging 3-7, 3-12
pausing during M-file execution 10-57
pcode 10-11
PCs
  MATLAB use of system resources 10-64
percent sign (comments) 10-9
performance
  improving for solvers 8-17
  obtaining statistics for ODE solvers 8-13
permission string 15-3
permutations 9-24
permute 12-2, 12-13
permuting array dimensions 12-13
  inverse 12-14
phase 6-36
pi 10-17
pi nv 4-23
pivoting
  partial 4-28
plane organization for structures 13-12
plotting
  mathematical functions 7-4
pol ar 10-5
poly 5-3
polyder 5-5
polyfit 5-6, 6-20, 6-23, 6-25, 6-27
polynomial
  fits to data 6-20
  interpolation 5-9
  regression 6-15
polynomials 5-1
  and curve fitting 5-6
  basic operations 5-2
  characteristics 5-3
  derivative of 5-5
  dividing 5-4
  evaluating in matrix sense 5-4
  multiplying 5-4
  representing 5-2
  roots 5-3
polyval 5-4, 6-23, 6-25, 6-27
positive definite factorization 4-26
power 10-22
power operator 10-22
powers
  matrix 4-33
preallocation 10-61
  cell array 10-61, 13-21
  structure array 10-61
precedence 10-22
  arithmetic operators 10-22
  object 14-16
  within expression 10-22
precision
  char 15-8
  double 15-8
  float 15-8
  for data types 15-8
  long 15-8
  short 15-8
  single 15-8
  uchar 15-8
preconditioner
  for sparse matrix 9-32
pre-processing data 6-12, 6-21
primary function 10-35
print 2-9
private directory 10-36
  and methods (object-oriented) 14-27
  in dispatching priority 10-10
private function 10-36
product
dot 4-8
inner 4-7
outer 4-7
profile
example 3-24
syntax 3-22
profiling 3-21
a function 3-22
chart of results 3-25
obtaining results 3-23
turning on and off 3-22
visualizing results 3-25
program
running external 10-58
programming
object-oriented 14-2
properties, ODE solvers 8-48
property structure (ODE)
creating 8-20
modifying 8-20
querying 8-21
pseudocode 10-11
pseudoinverses 4-23

Q
qr 4-30, 9-31
QR factorization 4-30, 9-31
quad 7-14, 8-47
quad8 7-14, 8-47
quadrature 7-14
questions and answers, ODE solvers
different kinds of systems 7-12
quick start
MATLAB programming 10-2

ODE solvers 8-3
quit 10-62
quitting
debug mode 3-20
quotes
for creating strings 11-4

R
rand 9-23
randn 12-6
rank 9-3
deficiency 4-31, 9-32
rational format 4-20
reading text separated by delimiters 2-38
real max 10-17
real min 10-17
reducing memory fragmentation 10-61
reference
subscripted 14-20
reference, passing arguments by 10-11
regression 6-15
linear-in-the-parameters 6-17
multiple 6-19
polynomial 6-15
relational operators 10-22
and empty arrays 10-24
and flow control 10-24
and strings 11-10
combining with others in expressions 10-28
precedence 10-22
relative accuracy (ODE) 8-21
remainder 10-13
removing
cells from cell array 13-23
fields from structure arrays 13-8
singleton dimensions 12-12
reorderings 9-24
   and LU factorization 9-28
   for sparser factorizations 9-26
   minimum degree ordering 9-27
   to reduce bandwidth 9-27
replacing substring within string 11-12
repmap 12-6
report of profiler results 3-23
representing
   polynomial roots 5-3
   polynomials 5-2
   problems for ODE solvers 8-5
reshape 12-11, 13-24
reshaping
   cell arrays 13-24
   multidimensional arrays 12-11
residuals 6-22
   for exponential data fit 6-27
residue 5-7
rigid body ODE example 8-33
rigidode 8-33
rmfield 13-8
roots
   of polynomial 5-3
root s 5-3
round 5-10
row vector 4-4
   for polynomial representation 5-2
rows
   deleting 10-39
running
   external program 10-58
   script 10-5
S
save 9-10, 10-62

saving
   ASCII files 2-23
   workspace contents 2-22
scalar 4-5
   and relational operators 11-10
   expansion 10-23
   string 11-10
schur 4-39
Schur decomposition 4-39
script 10-5
   and creation of new data 10-5
   and data in workspace 10-5
   characteristics 10-2
   defined 10-2
   example 10-5
   executing 10-5
   profile 3-22
script files 10-2
   for debugging 3-12
search path 2-26, 10-10
   and methods (object-oriented) 14-27
   and subfunctions 10-35
searching
   in string 11-12
   online help 2-32, 2-34
second difference operator, example 9-7
semicolon to suppress output 2-6
setfield 13-7
shadowed 2-26
shell escape functions 10-58
short 15-8
short integer 15-8
shortcuts
   for running MATLAB 2-2
simple inheritance 14-22
Simulink 1-8
Index

sin 10-5, 12-15
single precision 15-8
singular value decomposition 4-40
size
  of structure arrays 13-8
  of structure fields 13-8
size 9-23, 12-9, 13-8
smallest value system can represent 10-17
smoothing ODE solver output 8-24
solver
  See ODE solvers
solving linear systems of equations
  sparse 9-33
sort 9-27
sorting data 6-7
sparse 9-6, 9-24
sparse class 14-2
sparse data type 10-20
sparse matrix 10-19
  advantages 9-5
  and complex values 9-5
  Cholesky factorization 9-30
  computational considerations 9-23
  contents 9-11
  conversion from full 9-2, 9-6
  creating 9-6
    directly 9-7
    from diagonal elements 9-8
  defined 9-2
  density 9-6
  distance between nodes 9-21
eigenvalues 9-36
elementary 9-2
example 9-7
fill-in 9-20
importing 9-10
linear algebra 9-3
linear equations 9-3
linear systems of equations 9-33
LU factorization 9-28
  and reordering 9-28
mathematical operations 9-23
nonzero elements 9-11
  maximum number 9-7
  specifying when creating matrix 9-7
  storage 9-5, 9-11
  values 9-11
  nonzero elements of sparse matrix
    number of 9-11
  operations 9-23
  permutation 9-24
  preconditioner 9-32
  propagation through computations 9-23
  QR factorization 9-31
  reordering 9-3, 9-24
  storage 9-5
    for various permutations 9-26
    viewing 9-11
  theoretical graph 9-15
  triangular factorization 9-28
  viewing contents graphically 9-13
  visualizing 9-19
  working with 9-2
sparse ODE
  example 8-36
spconvert 9-10
spdiags 9-8
special values 10-17
speye 9-24, 9-26, 9-29
spones 9-27
spparms 9-27, 9-34
sprand 9-24
sprintf 15-16
spy 9-13
spy plot 9-19
sqrtm 4-33
square brackets
   for output arguments 10-7
squeeze 12-2, 12-12, 12-16
sscanf 15-14
stability (ODE solvers) 8-32
stack
   debugger 3-2, 3-8
starting MATLAB 2-2
start-up files 2-41
startup.m file 2-41
statements
   conditional 10-12
   continuing on multiple lines 2-6
statistics
   analyzing residuals 6-22
   correlation coefficients 6-9
   covariance 6-9
   descriptive 6-7
   pre-processing data 6-21
step
   in debugger 3-5
step size (ODE) 8-17, 8-28
   first step 8-29
   upper bound 8-28
stepping through code 3-16
stiff ODE
   example 8-36
stiffness (ODE), defined 8-8
storage
   array 10-39
   for various permutations of sparse matrix 9-26
   of sparse matrix 9-5
   sparse and full, comparison 9-11
   viewing for sparse matrix 9-11
strcmp 11-9
string 10-19
   and relational operators 11-10
array 11-5
   comparing values on cell arrays 11-10
   converting to cell arrays 11-7
   padding for equal row length 11-5
arrays
   cell array 11-7
   categorizing characters 11-11
   comparing 11-9
   concatenation 10-43
   conversion 11-3, 11-13, 15-2
   creating 11-4
   delimiting character 11-12
   evaluating 11-14
   evaluation 10-43
   finding starting position of substring 11-12
   functions that test 11-2
   operations 11-2
   removing trailing blanks 11-6
   representation 11-4
   scalar 11-10
   searching and replacing 11-12
   token 11-12
struct class 14-2
struct data type 10-20
structs 13-3, 13-5, 13-15
structure arrays
   accessing data 13-6
      for multiple fields 13-7
      using get field d 13-7
   adding fields 13-8
   applying functions to 13-8
   building 13-3
      using structs 13-5
   data organization 13-10
   defined 13-2
deleting fields 13-8
 element-by-element organization 13-13
 expanding 13-4, 13-5
 fields
 assigning data to 13-3
 assigning using setfield 13-7
 defined 13-3
 initializing with structs 13-5
 flat 13-16
 indexing
 nested structures 13-16
 within fields 13-6
 multidimensional 12-20
 applying functions 12-21
 nesting 13-15
 obtaining field names 13-4
 organizing data 13-10
 example 13-13
 overview 13-3
 plane organization 13-12
 preallocation 10-61
 size 13-8
 subscripting 13-4
 within cell arrays 13-31
 writing M-files for 13-9
 example 13-10
 structures
 used with classes 14-4
 structures. See structure arrays
 subarrays
 accessing 13-7
 subassign 14-21
 subfunctions 10-35
 accessing 10-35
 creating 10-35
 debugging 10-36
 function
 definition line 10-35
 in dispatching priority 10-10
 on search path 10-35
 subref 14-20
 subscripting
 how MATLAB calculates indices 10-42
 multidimensional arrays 12-3
 overloading 14-20
 structure arrays 13-4
 with logical expression 10-27
 with the find function 10-27
 subscripts 10-37
 page 12-3
 substring within a string 11-12
 subtraction
 of matrices 4-6
 subtraction operator 10-22
 sum 9-24, 9-27, 12-15
 super iort o 14-16
 superseding existing M-files names 10-36
 suppressing output 2-6
 surface plots
 to compare interpolation methods 5-14
 svd 4-41
 svds 9-37
 sw t ch 10-24, 10-31
 case groupings 10-31
 example 10-32
 multiple conditions 10-32
 otherwise condition 10-31
 Symbolic Math Toolbox 4-38
 sym m 9-27
 symmetric matrix 4-7
 sym mmd 9-29
 sym mcm9-27, 9-28, 9-29
 syntax highlighting 2-8
systems of equations. See linear systems of equations
systems of ODEs 7-12, 8-50

tabs in string arrays 11-11
tempdir 15-5
tempname 15-5
temporary files 15-2, 15-5
terminal events (ODE) 8-31
text editor. See editor
text files
delimiter-separated 2-38
The MathWorks
home page 1-8
theoretical graph 9-15
example 9-16
node 9-15
three-dimensional interpolation
nearest neighbor 5-15
tricubic 5-15
trilinear 5-15
time
interval (ODE) 8-6
numbers 10-51
steps, ODE solvers 8-47
token in string 11-12
tolerance 10-17
toolboxes 1-8
class definitions 14-2
creation
data type 10-19
transformed data
magnitude 6-36
phase 6-36
transforms 6-31
discrete Fourier 6-31
fast Fourier 6-31
fft 6-31
transpose 4-7
complex conjugate 4-8
unconjugated complex 4-8
transpose 12-14
transpose operator 10-22
triangular factorization
for sparse matrices 9-28
triangular matrices 4-26
triangulation 5-19
closest point searches 5-23
Delaunay 5-20
Voronoi diagrams 5-23
tricubic interpolation 5-15
trigonometric functions 10-5, 12-15
trilinear interpolation 5-15
troubleshooting, ODE solvers 8-49
tsearch 5-23
two-dimensional interpolation 5-12
bicubic 5-12
bilinear 5-12
nearest neighbor 5-12
uchar data type 15-8
uint data type 10-20
unary minus operator 10-22
unary plus operator 10-22
unconjugated complex transpose 4-8
unwrap 6-37
user input
obtaining interactively 10-57
UserObject data type 10-20
V
value
  largest system can represent 10-17
  passing arguments by 10-11
van der Pol example 8-34
  extra parameters 8-16
  simple, nonstiff 8-6
  simple, stiff 8-8
varargin 10-14, 13-26
  in argument list 10-15
  unpacking contents 10-14
varargout 10-15
  in argument list 10-15
  packing contents 10-15
variables
  containing filenames 2-24
  count 15-16
  deleting
    and memory use 10-62
  global 10-16
    rules for use 10-17
  in dispatching priority 10-10
  local 10-16
  memory usage 10-63
  naming 10-16
  replacing list with a cell array 13-24
  storage in memory 10-62
vdpode 8-34
vector
  column 4-4
  initial condition (ODE) 8-6
  of dates 10-54
  preallocation 10-61
  row 4-4
  time span vector (ODE) 8-11
vector products 4-7
vectorization 10-59
example 10-59
  for Jacobian matrix computation (ODE) 8-27
  replacing for loops 10-29
version 10-18
version, obtaining 10-18
visualizing
  cell array 13-20
  ODE solver results 8-7
  profiler results 3-25
  sparse matrix
    spy plot 9-19
voronoi 5-23
Voronoi diagrams 5-23

W
warnings 10-48
  displaying 10-49
Web site
  The MathWorks 1-8
while 10-24, 10-33
  and empty arrays 10-33
  example 10-33
  syntax 10-33
white space
  finding in string 11-11
whos 9-11, 12-9
  interpreting memory use 10-62
wildcards
  for load command 2-24
working environment
  MATLAB 1-4
workspace
  changing context in debugger 3-8, 3-12
  context 10-11
  loading data into 2-23
  of individual functions 10-11
saving contents 2-22

X
xor 10-26

Y
yellow arrow
  in the debugger 3-7

Z
zeros 9-23, 12-6