## Getting Started

### Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Overview</td>
<td>1-2</td>
</tr>
<tr>
<td>Overview of the MATLAB Environment</td>
<td>1-2</td>
</tr>
<tr>
<td>The MATLAB System</td>
<td>1-3</td>
</tr>
<tr>
<td>Documentation</td>
<td>1-5</td>
</tr>
<tr>
<td>Starting and Quitting the MATLAB Program</td>
<td>1-7</td>
</tr>
<tr>
<td>Starting a MATLAB Session</td>
<td>1-7</td>
</tr>
<tr>
<td>Quitting the MATLAB Program</td>
<td>1-8</td>
</tr>
</tbody>
</table>

### Matrices and Arrays

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrices and Magic Squares</td>
<td>2-2</td>
</tr>
<tr>
<td>About Matrices</td>
<td>2-2</td>
</tr>
<tr>
<td>Entering Matrices</td>
<td>2-4</td>
</tr>
<tr>
<td>sum, transpose, and diag</td>
<td>2-5</td>
</tr>
<tr>
<td>Subscripts</td>
<td>2-7</td>
</tr>
<tr>
<td>The Colon Operator</td>
<td>2-8</td>
</tr>
<tr>
<td>The magic Function</td>
<td>2-9</td>
</tr>
<tr>
<td>Expressions</td>
<td>2-11</td>
</tr>
<tr>
<td>Variables</td>
<td>2-11</td>
</tr>
<tr>
<td>Numbers</td>
<td>2-12</td>
</tr>
<tr>
<td>Operators</td>
<td>2-13</td>
</tr>
</tbody>
</table>
Choosing a Graph Type ........................................ 3-17
  Choosing a Type of Graph to Plot ....................... 3-17

Editing Plots ..................................................... 3-25
  Plot Edit Mode .............................................. 3-25
  Using Functions to Edit Graphs ......................... 3-30

Interactive Plotting .......................................... 3-31
  Plotting Two Variables with Plotting Tools ........... 3-31
  Changing the Appearance of Lines and Markers ........ 3-34
  Adding More Data to the Graph ......................... 3-35
  Changing the Type of Graph .............................. 3-38
  Modifying the Graph Data Source ....................... 3-40

Preparing Graphs for Presentation ....................... 3-45
  Annotating Graphs .................................... 3-45
  Printing the Graph .................................... 3-50
  Exporting the Graph ................................ 3-54

Basic Plotting Functions .................................. 3-58
  Creating a Plot ......................................... 3-58
  Plotting Multiple Data Sets in One Graph ............ 3-59
  Specifying Line Styles and Colors .................... 3-60
  Plotting Lines and Markers ............................. 3-61
  Graphing Imaginary and Complex Data .................. 3-63
  Adding Plots to an Existing Graph ..................... 3-64
  Figure Windows ....................................... 3-65
  Displaying Multiple Plots in One Figure .......... 3-66
  Controlling the Axes .................................. 3-68
  Adding Axis Labels and Titles ......................... 3-69
  Saving Figures ......................................... 3-70

Creating Mesh and Surface Plots ......................... 3-73
  About Mesh and Surface Plots ......................... 3-73
  Visualizing Functions of Two Variables ................ 3-73

Plotting Image Data ........................................ 3-81
  About Plotting Image Data .............................. 3-81
  Reading and Writing Images ............................ 3-82

Printing Graphics ........................................... 3-83
Overview of Printing ............................ 3-83  
Printing from the File Menu ........................ 3-83  
Exporting the Figure to a Graphics File ............. 3-84  
Using the Print Command ........................... 3-84  

**Understanding Handle Graphics Objects** ........................ 3-86  
Using the Handle ................................... 3-86  
Graphics Objects .................................... 3-87  
Setting Object Properties ........................... 3-89  
Specifying the Axes or Figure ......................... 3-92  
Finding the Handles of Existing Objects .......... 3-94  

---

### Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Control</td>
<td>4-2</td>
</tr>
<tr>
<td>Conditional Control — if, else, switch</td>
<td>4-2</td>
</tr>
<tr>
<td>Loop Control — for, while, continue, break</td>
<td>4-5</td>
</tr>
<tr>
<td>Error Control — try, catch</td>
<td>4-7</td>
</tr>
<tr>
<td>Program Termination — return</td>
<td>4-8</td>
</tr>
<tr>
<td>Other Data Structures</td>
<td>4-9</td>
</tr>
<tr>
<td>Multidimensional Arrays</td>
<td>4-9</td>
</tr>
<tr>
<td>Cell Arrays</td>
<td>4-11</td>
</tr>
<tr>
<td>Characters and Text</td>
<td>4-13</td>
</tr>
<tr>
<td>Structures</td>
<td>4-16</td>
</tr>
<tr>
<td>Scripts and Functions</td>
<td>4-20</td>
</tr>
<tr>
<td>Overview</td>
<td>4-20</td>
</tr>
<tr>
<td>Scripts</td>
<td>4-21</td>
</tr>
<tr>
<td>Functions</td>
<td>4-22</td>
</tr>
<tr>
<td>Types of Functions</td>
<td>4-24</td>
</tr>
<tr>
<td>Global Variables</td>
<td>4-26</td>
</tr>
<tr>
<td>Passing String Arguments to Functions</td>
<td>4-27</td>
</tr>
<tr>
<td>The eval Function</td>
<td>4-28</td>
</tr>
<tr>
<td>Function Handles</td>
<td>4-28</td>
</tr>
<tr>
<td>Function Functions</td>
<td>4-29</td>
</tr>
<tr>
<td>Vectorization</td>
<td>4-31</td>
</tr>
<tr>
<td>Preallocation</td>
<td>4-32</td>
</tr>
</tbody>
</table>
Matrices and Arrays

You can watch the Getting Started with MATLAB video demo for an overview of the major functionality.

- “Matrices and Magic Squares” on page 2-2
- “Expressions” on page 2-11
- “Working with Matrices” on page 2-17
- “More About Matrices and Arrays” on page 2-21
- “Controlling Command Window Input and Output” on page 2-31
Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example.

You can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions and save them in files.

Start by entering Dürer’s matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, ; , to indicate the end of each row.
- Surround the entire list of elements with square brackets, [ ].

To enter Dürer’s matrix, simply type in the Command Window

\[
A = \begin{bmatrix}
16 & 3 & 2 & 13 \\
5 & 10 & 11 & 8 \\
9 & 6 & 7 & 12 \\
4 & 15 & 14 & 1
\end{bmatrix}
\]
MATLAB displays the matrix you just entered:

\[
A = \begin{bmatrix}
16 & 3 & 2 & 13 \\
5 & 10 & 11 & 8 \\
9 & 6 & 7 & 12 \\
4 & 15 & 14 & 1 \\
\end{bmatrix}
\]

This matrix matches the numbers in the engraving. Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as \( A \). Now that you have \( A \) in the workspace, take a look at what makes it so interesting. Why is it magic?

**sum, transpose, and diag**

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is

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

MATLAB replies with

\[
\text{ans} = \begin{bmatrix}
34 & 34 & 34 & 34 \\
\end{bmatrix}
\]

When you do not specify an output variable, MATLAB uses the variable \text{ans}, short for \textit{answer}, to store the results of a calculation. You have computed a row vector containing the sums of the columns of \( A \). Each of the columns has the same sum, the \textit{magic} sum, 34.

How about the row sums? MATLAB has a preference for working with the columns of a matrix, so one way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. For an additional way that avoids the double transpose use the dimension argument for the \texttt{sum} function.

MATLAB has two transpose operators. The apostrophe operator (e.g., \( A' \)) performs a complex conjugate transposition. It flips a matrix about its main
diagonal, and also changes the sign of the imaginary component of any complex elements of the matrix. The dot-apostrophe operator (e.g., \( A' \)), transposes without affecting the sign of complex elements. For matrices containing all real elements, the two operators return the same result.

So

\[
A' 
\]

produces

\[
\begin{bmatrix}
16 & 5 & 9 & 4 \\
3 & 10 & 6 & 15 \\
2 & 11 & 7 & 14 \\
13 & 8 & 12 & 1
\end{bmatrix}
\]

and

\[
\text{sum}(A')'
\]

produces a column vector containing the row sums

\[
\begin{bmatrix}
34 \\
34 \\
34 \\
34 \\
34 \\
34
\end{bmatrix}
\]

The sum of the elements on the main diagonal is obtained with the \text{sum} and the \text{diag} functions:

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

produces

\[
\begin{bmatrix}
16 \\
10 \\
7 \\
1
\end{bmatrix}
\]
and

\[ \text{sum(diag(A))} \]

produces

\[ \text{ans} = \]
\[ 34 \]

The other diagonal, the so-called *antidiagonal*, is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, \texttt{flipud}, flips a matrix from left to right:

\[ \text{sum(diag(flipud(A)))} \]
\[ \text{ans} = \]
\[ 34 \]

You have verified that the matrix in Dürer’s engraving is indeed a magic square and, in the process, have sampled a few MATLAB matrix operations. The following sections continue to use this matrix to illustrate additional MATLAB capabilities.

**Subscripts**

The element in row \( i \) and column \( j \) of \( A \) is denoted by \( A(i,j) \). For example, \( A(4,2) \) is the number in the fourth row and second column. For the magic square, \( A(4,2) \) is 15. So to compute the sum of the elements in the fourth column of \( A \), type

\[ A(1,4) + A(2,4) + A(3,4) + A(4,4) \]

This subscript produces

\[ \text{ans} = \]
\[ 34 \]

but is not the most elegant way of summing a single column.

It is also possible to refer to the elements of a matrix with a single subscript, \( A(k) \). A single subscript is the usual way of referencing row and column vectors. However, 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 the 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:

```matlab
t = A(4,5)
Index exceeds matrix dimensions.
```

Conversely, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

```matlab
X = A;
X(4,5) = 17
```

\[
X = \\
\begin{bmatrix}
16 & 3 & 2 & 13 & 0 \\ \\
5 & 10 & 11 & 8 & 0 \\ \\
9 & 6 & 7 & 12 & 0 \\ \\
4 & 15 & 14 & 1 & 17
\end{bmatrix}
\]

**The Colon Operator**

The colon, `:`, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10:

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

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

is

```
100 93 86 79 72 65 58 51
```

and

```
0:pi/4:pi
```
is

\[
\begin{array}{cccc}
0 & 0.7854 & 1.5708 & 2.3562 \\
3.1416
\end{array}
\]

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 \( A \). Thus:

\[ \text{sum}(A(1:4,4)) \]

computes the sum of the fourth column. However, there is a better way to perform this computation. The colon by itself refers to all the elements in a row or column of a matrix and the keyword \texttt{end} refers to the last row or column. Thus:

\[ \text{sum}(A(:,\text{end})) \]

computes the sum of the elements in the last column of \( A \):

\[
\text{ans} = \\
34
\]

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

\[ \frac{\text{sum}(1:16)}{4} \]

which, of course, is

\[
\text{ans} = \\
34
\]

**The magic Function**

MATLAB actually has a built-in function that creates magic squares of almost any size. Not surprisingly, this function is named \texttt{magic}:  

\[
\begin{array}{cccc}
0 & 0.7854 & 1.5708 & 2.3562 \\
3.1416
\end{array}
\]
\[
B = \text{magic}(4)
\]
\[
B =
\begin{bmatrix}
16 &  2 &  3 & 13 \\
 5 & 11 & 10 &  8 \\
 9 &  7 &  6 & 12 \\
 4 & 14 & 15 &  1 \\
\end{bmatrix}
\]

This matrix is almost the same as the one in the Dürer engraving and has all the same “magic” properties; the only difference is that the two middle columns are exchanged.

To make this \( B \) into Dürer’s \( A \), swap the two middle columns:

\[
A = B(:,[1\ 3\ 2\ 4])
\]

This subscript indicates that—for each of the rows of matrix \( B \)—reorder the elements in the order 1, 3, 2, 4. It produces:

\[
A =
\begin{bmatrix}
16 &  3 &  2 & 13 \\
 5 & 10 & 11 &  8 \\
 9 &  6 &  7 & 12 \\
 4 & 15 & 14 &  1 \\
\end{bmatrix}
\]
Working with Matrices

In this section...

“Generating Matrices” on page 2-17
“The load Function” on page 2-18
“Saving Code to a File” on page 2-18
“Concatenation” on page 2-19
“Deleting Rows and Columns” on page 2-20

Generating Matrices

MATLAB software provides four functions that generate basic matrices.

zeros All zeros
ones All ones
rand Uniformly distributed random elements
randn Normally distributed random elements

Here are some examples:

\[
Z = \text{zeros}(2,4)
\]
\[
Z = \\
0 0 0 0 \\
0 0 0 0
\]

\[
F = 5*\text{ones}(3,3)
\]
\[
F = \\
5 5 5 \\
5 5 5 \\
5 5 5
\]

\[
N = \text{fix}(10*\text{rand}(1,10))
\]
\[
N = \\
9 2 6 4 8 7 4 0 8 4
\]
R = randn(4,4)
R =
    0.6353  0.0860  -0.3210  -1.2316
    -0.6014 -2.0046   1.2366   1.0556
     0.5512 -0.4931  -0.6313  -0.1132
    -1.0998  0.4620  -2.3252  0.3792

The load Function
The load function reads binary files containing matrices generated by earlier MATLAB sessions, or reads text files containing numeric data. The text file should be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row. For example, outside of MATLAB, create a text file containing these four lines:

```
16.0  3.0  2.0  13.0
  5.0 10.0 11.0  8.0
  9.0  6.0  7.0  12.0
  4.0 15.0 14.0  1.0
```

Save the file as `magik.dat` in the current directory. The statement

```
load magik.dat
```

reads the file and creates a variable, `magik`, containing the example matrix.

An easy way to read data into MATLAB from many text or binary formats is to use the Import Wizard.

Saving Code to a File
You can create matrices using text files containing MATLAB code. Use the MATLAB Editor or another text editor to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in `.m`.

For example, create a file in the current directory named `magik.m` containing these five lines:
A = [16.0 3.0 2.0 13.0
     5.0 10.0 11.0 8.0
     9.0 6.0 7.0 12.0
     4.0 15.0 14.0 1.0 ];

The statement

magik

reads the file and creates a variable, A, containing the example matrix.

**Concatenation**

*Concatenation* is the process of joining small matrices 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:

\[
B =
\begin{bmatrix}
16 & 3 & 2 & 13 & 48 & 35 & 34 & 45 \\
5 & 10 & 11 & 8 & 37 & 42 & 43 & 40 \\
9 & 6 & 7 & 12 & 41 & 38 & 39 & 44 \\
4 & 15 & 14 & 1 & 36 & 47 & 46 & 33 \\
64 & 51 & 50 & 61 & 32 & 19 & 18 & 29 \\
53 & 58 & 59 & 56 & 21 & 26 & 27 & 24 \\
57 & 54 & 55 & 60 & 25 & 22 & 23 & 28 \\
52 & 63 & 62 & 49 & 20 & 31 & 30 & 17
\end{bmatrix}
\]

This matrix is halfway 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:

\[
\text{sum}(B)
\]

\[
\text{ans} =
\begin{bmatrix}
\end{bmatrix}
\]
But its row sums, \( \text{sum}(B')' \), 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

\[
X =
\begin{pmatrix}
16 & 2 & 13 \\
5 & 11 & 8 \\
9 & 7 & 12 \\
4 & 14 & 1
\end{pmatrix}
\]

If you delete a single element from a matrix, the result is not 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

\[
X =
\begin{pmatrix}
16 & 9 & 2 & 7 & 13 & 12 & 1
\end{pmatrix}
\]
More About Matrices and Arrays

In this section...

“Linear Algebra” on page 2-21
“Arrays” on page 2-25
“Multivariate Data” on page 2-27
“Scalar Expansion” on page 2-28
“Logical Subscripting” on page 2-28
“The find Function” on page 2-29

Linear Algebra
Informally, the terms *matrix* and *array* are often used interchangeably. More precisely, a *matrix* is a two-dimensional numeric array that represents a linear transformation. The mathematical operations defined on matrices are the subject of linear algebra.

Dürer’s magic square

\[
A = \begin{bmatrix}
16 & 3 & 2 & 13 \\
5 & 10 & 11 & 8 \\
9 & 6 & 7 & 12 \\
4 & 15 & 14 & 1 \\
\end{bmatrix}
\]

provides several examples that give a taste of MATLAB matrix operations. You have already seen the matrix transpose, \(A'\). Adding a matrix to its transpose produces a *symmetric* matrix:

\[
A + A'
\]

\[
\text{ans} =
\begin{bmatrix}
32 & 8 & 11 & 17 \\
8 & 20 & 17 & 23 \\
11 & 17 & 14 & 26 \\
17 & 23 & 26 & 2 \\
\end{bmatrix}
\]
The multiplication symbol, *, denotes the matrix multiplication involving inner products between rows and columns. Multiplying the transpose of a matrix by the original matrix also produces a symmetric matrix:

$$A^T A$$

$$\begin{array}{llll}
378 & 212 & 206 & 360 \\
212 & 370 & 368 & 206 \\
206 & 368 & 370 & 212 \\
360 & 206 & 212 & 378
\end{array}$$

The determinant of this particular matrix happens to be zero, indicating that the matrix is singular:

$$d = \text{det}(A)$$

$$d = 0$$

The reduced row echelon form of A is not the identity:

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

$$\begin{array}{llll}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & -3 \\
0 & 0 & 1 & 3 \\
0 & 0 & 0 & 0
\end{array}$$

Because the matrix is singular, it does not have an inverse. If you try to compute the inverse with

$$X = \text{inv}(A)$$

you will get a warning message:

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 9.796086e-018.
```

Roundoff error has prevented the matrix inversion algorithm from detecting exact singularity. But the value of rcond, which stands for reciprocal
condition estimate, is on the order of \texttt{eps}, the floating-point relative precision, so the computed inverse is unlikely to be of much use.

The eigenvalues of the magic square are interesting:

\begin{verbatim}
   e = eig(A)

   e =
       34.0000
       8.0000
       0.0000
      -8.0000
\end{verbatim}

One of the eigenvalues is zero, which is another consequence of singularity. The largest eigenvalue is 34, the magic sum. That sum results because the vector of all ones is an eigenvector:

\begin{verbatim}
   v = ones(4,1)

   v =
       1
       1
       1
       1

   A*v

   ans =
       34
       34
       34
       34
       34
\end{verbatim}

When a magic square is scaled by its magic sum,

\begin{verbatim}
   P = A/34
\end{verbatim}

the result is a doubly stochastic matrix whose row and column sums are all 1:
\[
P = \\
0.4706 & 0.0882 & 0.0588 & 0.3824 \\
0.1471 & 0.2941 & 0.3235 & 0.2353 \\
0.2647 & 0.1765 & 0.2059 & 0.3529 \\
0.1176 & 0.4412 & 0.4118 & 0.0294 \\
\]

Such matrices represent the transition probabilities in a *Markov process*. Repeated powers of the matrix represent repeated steps of the process. For this example, the fifth power

\[
P^5 \\
0.2507 & 0.2495 & 0.2494 & 0.2504 \\
0.2497 & 0.2501 & 0.2502 & 0.2500 \\
0.2500 & 0.2498 & 0.2499 & 0.2503 \\
0.2496 & 0.2506 & 0.2505 & 0.2493 \\
\]

This shows that as \( k \) approaches infinity, all the elements in the \( k \)th power, \( P^k \), approach \( \frac{1}{4} \).

Finally, the coefficients in the characteristic polynomial

\[
\text{poly}(A) \\
1 & -34 & -64 & 2176 & 0 \\
\]

These coefficients indicate that the characteristic polynomial \( \det(A - \lambda I) \)

is

\[
\lambda^4 - 34\lambda^3 - 64\lambda^2 + 2176\lambda \\
\]

The constant term is zero because the matrix is singular. The coefficient of the cubic term is -34 because the matrix is magic!
Arrays

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are done element by element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

The list of operators includes

- Addition
- Subtraction
- Element-by-element multiplication
- Element-by-element division
- Element-by-element left division
- Element-by-element power
- Unconjugated array transpose

If the Dürer magic square is multiplied by itself with array multiplication

\[ A \cdot A \]

the result is an array containing the squares of the integers from 1 to 16, in an unusual order:

\[
\begin{array}{cccc}
256 & 9 & 4 & 169 \\
25 & 100 & 121 & 64 \\
81 & 36 & 49 & 144 \\
16 & 225 & 196 & 1
\end{array}
\]

Building Tables

Array operations are useful for building tables. Suppose \( n \) is the column vector

\[ n = (0:9)'; \]
Then

\[
pows = [n \ n.^2 \ 2.^n]
\]

builds a table of squares and powers of 2:

\[
pows =
\begin{array}{ccc}
0 & 0 & 1 \\
1 & 1 & 2 \\
2 & 4 & 4 \\
3 & 9 & 8 \\
4 & 16 & 16 \\
5 & 25 & 32 \\
6 & 36 & 64 \\
7 & 49 & 128 \\
8 & 64 & 256 \\
9 & 81 & 512 \\
\end{array}
\]

The elementary math functions operate on arrays element by element. So

\[
format short g
x = (1:0.1:2)';
logs = [x log10(x)]
\]

builds a table of logarithms.

\[
logs =
\begin{array}{cc}
1.0 & 0 \\
1.1 & 0.04139 \\
1.2 & 0.07918 \\
1.3 & 0.11394 \\
1.4 & 0.14613 \\
1.5 & 0.17609 \\
1.6 & 0.20412 \\
1.7 & 0.23045 \\
1.8 & 0.25527 \\
1.9 & 0.27875 \\
2.0 & 0.30103 \\
\end{array}
\]
**Multivariate Data**

MATLAB uses column-oriented analysis for multivariate statistical data. Each column in a data set represents a variable and each row an observation. The \((i,j)\)th element is the \(i\)th observation of the \(j\)th variable.

As an example, consider a data set with three variables:

- Heart rate
- Weight
- Hours of exercise per week

For five observations, the resulting array might look like

\[
D = \begin{bmatrix}
72 & 134 & 3.2 \\
81 & 201 & 3.5 \\
69 & 156 & 7.1 \\
82 & 148 & 2.4 \\
75 & 170 & 1.2
\end{bmatrix}
\]

The first row contains the heart rate, weight, and exercise hours for patient 1, the second row contains the data for patient 2, and so on. Now you can apply many MATLAB data analysis functions to this data set. For example, to obtain the mean and standard deviation of each column, use

\[
\text{mu} = \text{mean}(D), \quad \text{sigma} = \text{std}(D)
\]

\[
\text{mu} =
\begin{bmatrix}
75.8 \\
161.8 \\
3.48
\end{bmatrix}
\]

\[
\text{sigma} =
\begin{bmatrix}
5.6303 \\
25.499 \\
2.2107
\end{bmatrix}
\]

For a list of the data analysis functions available in MATLAB, type

\[
\text{help datafun}
\]

If you have access to the Statistics Toolbox™ software, type

\[
\text{help stats}
\]
**Scalar Expansion**

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in our magic square is 8.5, so

\[
B = A - 8.5
\]

forms a matrix whose column sums are zero:

\[
B = \begin{bmatrix}
7.5 & -5.5 & -6.5 & 4.5 \\
-3.5 & 1.5 & 2.5 & -0.5 \\
0.5 & -2.5 & -1.5 & 3.5 \\
-4.5 & 6.5 & 5.5 & -7.5
\end{bmatrix}
\]

\[
\text{sum}(B)
\]

\[
\text{ans} = 
\begin{bmatrix}
0 & 0 & 0 & 0
\end{bmatrix}
\]

With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example,

\[
B(1:2,2:3) = 0
\]

zeroes out a portion of \(B\):

\[
B = \begin{bmatrix}
7.5 & 0 & 0 & 4.5 \\
-3.5 & 0 & 0 & -0.5 \\
0.5 & -2.5 & -1.5 & 3.5 \\
-4.5 & 6.5 & 5.5 & -7.5
\end{bmatrix}
\]

**Logical Subscripting**

The logical vectors created from logical and relational operations can be used to reference subarrays. Suppose \(X\) is an ordinary matrix and \(L\) is a matrix of the same size that is the result of some logical operation. Then \(X(L)\) specifies the elements of \(X\) where the elements of \(L\) are nonzero.
This kind of subscripting can be done in one step by specifying the logical operation as the subscripting expression. Suppose you have the following set of data:

\[
x = [2.1 \ 1.7 \ 1.6 \ 1.5 \ \text{NaN} \ 1.9 \ 1.8 \ 1.5 \ 1.8 \ 1.4 \ 2.2 \ 1.6 \ 1.8];
\]

The NaN is a marker for a missing observation, such as a failure to respond to an item on a questionnaire. To remove the missing data with logical indexing, use \text{isfinite}(x), which is true for all finite numerical values and false for NaN and Inf:

\[
x = x(\text{isfinite}(x))
\]

\[
x = 2.1 \ 1.7 \ 1.6 \ 1.5 \ 1.9 \ 1.8 \ 1.5 \ 1.8 \ 1.4 \ 2.2 \ 1.6 \ 1.8
\]

Now there is one observation, 5.1, which seems to be very different from the others. It is an outlier. The following statement removes outliers, in this case those elements more than three standard deviations from the mean:

\[
x = x(\text{abs}(x-\text{mean}(x)) \leq 3*\text{std}(x))
\]

\[
x = 2.1 \ 1.7 \ 1.6 \ 1.5 \ 1.9 \ 1.8 \ 1.5 \ 1.8 \ 1.4 \ 2.2 \ 1.6 \ 1.8
\]

For another example, highlight the location of the prime numbers in Dürer’s magic square by using logical indexing and scalar expansion to set the nonprimes to 0. (See “The magic Function” on page 2-9.)

\[
A(-\text{isprime}(A)) = 0
\]

\[
A =
\begin{bmatrix}
0 & 3 & 2 & 13 \\
5 & 0 & 11 & 0 \\
0 & 0 & 7 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

**The find Function**

The \text{find} function determines the indices of array elements that meet a given logical condition. In its simplest form, \text{find} returns a column vector of indices. Transpose that vector to obtain a row vector of indices. For example, start again with Dürer’s magic square. (See “The magic Function” on page 2-9.)
\[ k = \text{find}(\text{isprime}(A))' \]

picks out the locations, using one-dimensional indexing, of the primes in the magic square:

\[
\begin{array}{ccccccc}
2 & 5 & 9 & 10 & 11 & 13 \\
\end{array}
\]

Display those primes, as a row vector in the order determined by \( k \), with

\[ A(k) \]

\[
\begin{array}{ccccccc}
5 & 3 & 2 & 11 & 7 & 13 \\
\end{array}
\]

When you use \( k \) as a left-hand-side index in an assignment statement, the matrix structure is preserved:

\[ A(k) = \text{NaN} \]

\[
\begin{array}{ccccccc}
16 & \text{NaN} & \text{NaN} & \text{NaN} \\
\text{NaN} & 10 & \text{NaN} & 8 \\
9 & 6 & \text{NaN} & 12 \\
4 & 15 & 14 & 1 \\
\end{array}
\]
Graphics

- “Plotting Data” on page 3-2
- “Plotting Techniques” on page 3-3
- “Graph Components” on page 3-7
- “Choosing a Graph Type” on page 3-17
- “Editing Plots” on page 3-25
- “Interactive Plotting” on page 3-31
- “Preparing Graphs for Presentation” on page 3-45
- “Basic Plotting Functions” on page 3-58
- “Creating Mesh and Surface Plots” on page 3-73
- “Plotting Image Data” on page 3-81
- “Printing Graphics” on page 3-83
- “Understanding Handle Graphics Objects” on page 3-86
Basic Plotting Functions

In this section...

“Creating a Plot” on page 3-58
“Plotting Multiple Data Sets in One Graph” on page 3-59
“Specifying Line Styles and Colors” on page 3-60
“Plotting Lines and Markers” on page 3-61
“Graphing Imaginary and Complex Data” on page 3-63
“Adding Plots to an Existing Graph” on page 3-64
“Figure Windows” on page 3-65
“Displaying Multiple Plots in One Figure” on page 3-66
“Controlling the Axes” on page 3-68
“Adding Axis Labels and Titles” on page 3-69
“Saving Figures” on page 3-70

Creating a Plot

The plot function has different forms, depending on the input arguments. If \( y \) is a vector, \( \text{plot}(y) \) produces a piecewise linear graph of the elements of \( y \) versus the index of the elements of \( y \). If you specify two vectors as arguments, \( \text{plot}(x,y) \) produces a graph of \( y \) versus \( x \).

For example, these statements use the colon operator to create a vector of \( x \) values ranging from 0 to \( 2\pi \), compute the sine of these values, and plot the result:

\[
\begin{align*}
x &= 0:pi/100:2*pi; \\
y &= \sin(x); \\
\text{plot}(x,y)
\end{align*}
\]

Now label the axes and add a title. The characters \( \pi \) create the symbol \( \pi \).

See “text strings” in the MATLAB Reference documentation for more symbols:

\[
\begin{align*}
xlabel('x = 0:2\pi') \\
ylabel('Sine of x')
\end{align*}
\]
Plotting Multiple Data Sets in One Graph

Multiple x-y pair arguments create multiple graphs with a single call to plot. automatically cycle through a predefined (but customizable) list of colors to allow discrimination among sets of data. See the axes ColorOrder andLineStyleOrder properties.

For example, these statements plot three related functions of x, with each curve in a separate distinguishing color:

```matlab
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y,x,y2,x,y3)
```

The legend command provides an easy way to identify the individual plots:

Specifying Line Styles and Colors
It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command:

\[
\text{plot}(x,y,'color\_style\_marker')
\]

\textit{color\_style\_marker} is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type. The strings are composed of combinations of the following elements:
<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td></td>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td></td>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td>'w'</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>Line style</td>
<td>'- '</td>
<td>solid</td>
</tr>
<tr>
<td></td>
<td>'-- '</td>
<td>dashed</td>
</tr>
<tr>
<td></td>
<td>':'</td>
<td>dotted</td>
</tr>
<tr>
<td></td>
<td>'.-'</td>
<td>dash-dot</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>no line</td>
</tr>
<tr>
<td>Marker type</td>
<td>'+'</td>
<td>plus mark</td>
</tr>
<tr>
<td></td>
<td>'o'</td>
<td>unfilled circle</td>
</tr>
<tr>
<td></td>
<td>'*'</td>
<td>asterisk</td>
</tr>
<tr>
<td></td>
<td>'x'</td>
<td>letter x</td>
</tr>
<tr>
<td></td>
<td>'s'</td>
<td>filled square</td>
</tr>
<tr>
<td></td>
<td>'d'</td>
<td>filled diamond</td>
</tr>
<tr>
<td></td>
<td>'^'</td>
<td>filled upward triangle</td>
</tr>
<tr>
<td></td>
<td>'v'</td>
<td>filled downward triangle</td>
</tr>
<tr>
<td></td>
<td>'&gt;'</td>
<td>filled right-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'&lt;'</td>
<td>filled left-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'p'</td>
<td>filled pentagram</td>
</tr>
<tr>
<td></td>
<td>'h'</td>
<td>filled hexagram</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>no marker</td>
</tr>
</tbody>
</table>

You can also edit color, line style, and markers interactively. See “Editing Plots” on page 3-25 for more information.

**Plotting Lines and Markers**

If you specify a marker type but not a line style, only the marker is drawn. For example,
\begin{verbatim}
plot(x,y,'ks')
\end{verbatim}

plots black squares at each data point, but does not connect the markers with a line.

The statement
\begin{verbatim}
plot(x,y,'r:+')
\end{verbatim}

plots a red-dotted line and places plus sign markers at each data point.

**Placing Markers at Every Tenth Data Point**

You might want to use fewer data points to plot the markers than you use to plot the lines. This example plots the data twice using a different number of points for the dotted line and marker plots:

\begin{verbatim}
x1 = 0:pi/100:2*pi;
x2 = 0:pi/10:2*pi;
plot(x1,sin(x1),'r:'),x2,sin(x2),'r+')
\end{verbatim}
Graphing Imaginary and Complex Data

When the arguments to `plot` are complex, the imaginary part is ignored except when you pass `plot` a single complex argument. For this special case, the command is a shortcut for a graph of the real part versus the imaginary part. Therefore,

```
plot(Z)
```

where `Z` is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

For example,

```
t = 0:pi/10:2*pi;
plot(exp(i*t),'-o')
axis equal
```

draws a 20-sided polygon with little circles at the vertices. The `axis equal` command makes the individual tick-mark increments on the x- and y-axes the same length, which makes this plot more circular in appearance.
Adding Plots to an Existing Graph

The MATLAB `hold` command enables you to add plots to an existing graph. When you type

```
hold on
```

Now MATLAB does not replace the existing graph when you issue another plotting command; it adds the new data to the current graph, rescaling the axes if necessary.

For example, these statements first create a contour plot of the `peaks` function, then superimpose a pseudocolor plot of the same function:

```
[x,y,z] = peaks;
pcolor(x,y,z)
shading interp
hold on
contour(x,y,z,20,'k')
hold off
```

The `hold on` command combines the `pcolor` plot with the `contour` plot in one figure.
Programming

If you have an active Internet Explorer® connection, you can watch the Writing a MATLAB Program video demo for an overview of the major functionality.

- “Flow Control” on page 4-2
- “Other Data Structures” on page 4-9
- “Scripts and Functions” on page 4-20
- “Object-Oriented Programming” on page 4-33
Flow Control

In this section...

| Conditional Control — if, else, switch” on page 4-2 |
| “Loop Control — for, while, continue, break” on page 4-5 |
| “Error Control — try, catch” on page 4-7 |
| “Program Termination — return” on page 4-8 |

Conditional Control — if, else, switch
Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an if statement. For example:

```matlab
% Generate a random number
a = randi(100, 1);

% If it is even, divide by 2
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

If statements can include alternate choices, using the optional keywords elseif or else. For example:

```matlab
a = randi(100, 1);

if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end
```

Alternatively, when you want to test for equality against a set of known values, use a switch statement. For example:
[dayNum, dayString] = weekday(date, 'long', 'en_US');

switch dayString
    case 'Monday'
        disp('Start of the work week')
    case 'Tuesday'
        disp('Day 2')
    case 'Wednesday'
        disp('Day 3')
    case 'Thursday'
        disp('Day 4')
    case 'Friday'
        disp('Last day of the work week')
    otherwise
        disp('Weekend!')
end

For both if and switch, MATLAB executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the end keyword.

In general, when you have many possible discrete, known values, switch statements are easier to read than if statements. However, you cannot test for inequality between switch and case values. For example, you cannot implement this type of condition with a switch:

yourNumber = input('Enter a number: ');

if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end

**Array Comparisons in Conditional Statements**

It is important to understand how relational operators and if statements work with matrices. When you want to check for equality between two variables, you might use
if A == B, ...

This is valid MATLAB code, and does what you expect when A and B are scalars. But when A and B are matrices, A == B does not test if they are equal, it tests where they are equal; the result is another matrix of 0's and 1's showing element-by-element equality. (In fact, if A and B are not the same size, then A == B is an error.)

```matlab
A = magic(4); B = A; B(1,1) = 0;
```

```matlab
A == B
ans =
 0 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
```

The proper way to check for equality between two variables is to use the `isequal` function:

```matlab
if isequal(A, B), ...
```

`isequal` returns a scalar logical value of 1 (representing true) or 0 (false), instead of a matrix, as the expression to be evaluated by the `if` function.

Using the A and B matrices from above, you get

```matlab
isequal(A, B)
an =
 0
```

Here is another example to emphasize this point. If A and B are scalars, the following program will never reach the “unexpected situation”. But for most pairs of matrices, including our magic squares with interchanged columns, none of the matrix conditions A > B, A < B, or A == B is true for all elements and so the else clause is executed:

```matlab
if A > B
'greater'
elseif A < B
'less'
elseif A == B
'equal'
```
else
    error('Unexpected situation')
end

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with if, including

    isequal
    isempty
    all
    any

Loop Control — for, while, continue, break

This section covers those MATLAB functions that provide control over program loops.

for

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delineates the statements:

```matlab
for n = 3:32
    r(n) = rank(magic(n));
end
r
```

The semicolon terminating the inner statement suppresses repeated printing, and the r after the loop displays the final result.

It is a good idea to indent the loops for readability, especially when they are nested:

```matlab
for i = 1:m
    for j = 1:n
        H(i,j) = 1/(i+j);
    end
end
```
**while**

The `while` loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching `end` delineates the statements.

Here is a complete program, illustrating `while`, `if`, `else`, and `end`, that uses interval bisection to find a zero of a polynomial:

```plaintext
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

The result is a root of the polynomial $x^3 - 2x - 5$, namely

$$x = 2.09455148154233$$

The cautions involving matrix comparisons that are discussed in the section on the `if` statement also apply to the `while` statement.

**continue**

The `continue` statement passes control to the next iteration of the `for` loop or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered:
% fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);

break

The `break` statement lets you exit early from a `for` loop or `while` loop. In nested loops, `break` exits from the innermost loop only.

Here is an improvement on the example from the previous section. Why is this use of `break` a good idea?

```matlab
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if fx == 0
        break
    elseif sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

Error Control — try, catch

This section covers those MATLAB functions that provide error handling control.
try
The general form of a try-catch statement sequence is

```matlab
try
  statement
  ...
  statement
catch exceptObj
  statement
  ...
  statement
end
```

In this sequence, the statements in the try block (that part of the try-catch that follows the word try statement, and precedes catch) between try and catch execute just like any other program code. If an error occurs within the try section the statements between catch and end are then executed. Examine the contents of the MException object exceptObj to see the cause of the error. If an error occurs between catch and end, MATLAB terminates execution unless another try-catch sequence has been established.

Program Termination — return
This section covers the MATLAB return function that enables you to terminate your program before it runs to completion.

```matlab
return
```

return terminates the current sequence of commands and returns control to the invoking function or to the keyboard. return is also used to terminate keyboard mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a return statement within the called function to force an early termination and to transfer control to the invoking function.
Other Data Structures

Other Data Structures

In this section...

“Multidimensional Arrays” on page 4-9
“Cell Arrays” on page 4-11
“Characters and Text” on page 4-13
“Structures” on page 4-16

Multidimensional Arrays

Multidimensional arrays in the MATLAB environment are arrays with more than two subscripts. One way of creating a multidimensional array is by calling zeros, ones, rand, or randn with more than two arguments. For example,

\[ R = \text{randn}(3,4,5); \]

creates a 3-by-4-by-5 array with a total of \(3 \times 4 \times 5 = 60\) normally distributed random elements.

A three-dimensional array might represent three-dimensional physical data, say the temperature in a room, sampled on a rectangular grid. Or it might represent a sequence of matrices, \(A^{(k)}\), or samples of a time-dependent matrix, \(A(t)\). In these latter cases, the \((i, j)\)th element of the \(k\)th matrix, or the \(t_{k}\)th matrix, is denoted by \(A(i,j,k)\).

MATLAB and Dürer’s versions of the magic square of order 4 differ by an interchange of two columns. Many different magic squares can be generated by interchanging columns. The statement

\[ p = \text{perms}(1:4); \]

generates the \(4! = 24\) permutations of \(1:4\). The \(k\)th permutation is the row vector \(p(k,:\)). Then

\[ A = \text{magic}(4); \]
\[ M = \text{zeros}(4,4,24); \]
for k = 1:24
    M(:,:,k) = A(:,p(k,:));
end

stores the sequence of 24 magic squares in a three-dimensional array, M. The size of M is

size(M)

ans =
    4     4    24

Note The order of the matrices shown in this illustration might differ from your results. The perms function always returns all permutations of the input vector, but the order of the permutations might be different for different MATLAB versions.

The statement

    sum(M,d)

computes sums by varying the dth subscript. So

    sum(M,1)
Other Data Structures

is a 1-by-4-by-24 array containing 24 copies of the row vector

\[
\begin{array}{cccc}
34 & 34 & 34 & 34 \\
\end{array}
\]

and

\[
\text{sum}(M,2)
\]

is a 4-by-1-by-24 array containing 24 copies of the column vector

\[
\begin{array}{cccc}
34 \\
34 \\
34 \\
34 \\
\end{array}
\]

Finally,

\[
S = \text{sum}(M,3)
\]

adds the 24 matrices in the sequence. The result has size 4-by-4-by-1, so it looks like a 4-by-4 array:

\[
\begin{array}{cccc}
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
\end{array}
\]

**Cell Arrays**

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example,

\[
C = \{A \text{ sum}(A) \text{ prod(prod(A))}\}
\]

produces a 1-by-3 cell array. The three cells contain the magic square, the row vector of column sums, and the product of all its elements. When \(C\) is displayed, you see

\[
C =
\]
This is because the first two cells are too large to print in this limited space, but the third cell contains only a single number, 16!, so there is room to print it.

Here are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces. For example, \( C\{1\} \) retrieves the magic square and \( C\{3\} \) is 16!. Second, cell arrays contain copies of other arrays, not pointers to those arrays. If you subsequently change \( A \), nothing happens to \( C \).

You can use three-dimensional arrays to store a sequence of matrices of the same size. Cell arrays can be used to store a sequence of matrices of different sizes. For example,

\[
M = \text{cell}(8,1);
\text{for } n = 1:8
\quad M\{n\} = \text{magic}(n);
\text{end}
\]

\( M \)

produces a sequence of magic squares of different order:

\[
M = \\
[ \quad 1 ] \\
[ 2x2 \text{ double} ] \\
[ 3x3 \text{ double} ] \\
[ 4x4 \text{ double} ] \\
[ 5x5 \text{ double} ] \\
[ 6x6 \text{ double} ] \\
[ 7x7 \text{ double} ] \\
[ 8x8 \text{ double} ]
\]
You can retrieve the 4-by-4 magic square matrix with

\[ \text{M}\{4\} \]

**Characters and Text**

Enter text into MATLAB using single quotes. For example,

\[ s = 'Hello' \]

The result is not the same kind of numeric matrix or array you have been dealing with up to now. It is a 1-by-5 character array.
Internally, the characters are stored as numbers, but not in floating-point format. The statement

\[ a = \text{double}(s) \]

converts the character array to a numeric matrix containing floating-point representations of the ASCII codes for each character. The result is

\[ a = \begin{bmatrix} 72 & 101 & 108 & 108 & 111 \end{bmatrix} \]

The statement

\[ s = \text{char}(a) \]

reverses the conversion.

Converting numbers to characters makes it possible to investigate the various fonts available on your computer. The printable characters in the basic ASCII character set are represented by the integers \(32:127\). (The integers less than 32 represent nonprintable control characters.) These integers are arranged in an appropriate 6-by-16 array with

\[ F = \text{reshape}(32:127,16,6)'; \]

The printable characters in the extended ASCII character set are represented by \(F+128\). When these integers are interpreted as characters, the result depends on the font currently being used. Type the statements

\[
\begin{align*}
\text{char}(F) \\
\text{char}(F+128)
\end{align*}
\]

and then vary the font being used for the Command Window. Select **Preferences** from the **File** menu and then click **Fonts** to change the font. If you include tabs in lines of code, use a fixed-width font, such as Monospaced, to align the tab positions on different lines.

Concatenation with square brackets joins text variables together into larger strings. The statement

\[ h = [s, ' world'] \]
joins the strings horizontally and produces

\[
    h = \\
    \text{Hello world}
\]

The statement

\[
    v = [s; 'world']
\]

joins the strings vertically and produces

\[
    v = \\
    \text{Hello} \\
    \text{world}
\]

Note that a blank has to be inserted before the 'w' in h and that both words in v have to have the same length. The resulting arrays are both character arrays; h is 1-by-11 and v is 2-by-5.

To manipulate a body of text containing lines of different lengths, you have two choices—a padded character array or a cell array of strings. When creating a character array, you must make each row of the array the same length. (Pad the ends of the shorter rows with spaces.) The char function does this padding for you. For example,

\[
    S = \text{char('A','rolling','stone','gathers','momentum.')}
\]

produces a 5-by-9 character array:

\[
    S = \\
    A \\
    \text{rolling} \\
    \text{stone} \\
    \text{gathers} \\
    \text{momentum.}
\]

Alternatively, you can store the text in a cell array. For example,

\[
    C = \{'A';'rolling';'stone';'gathers';'momentum.'\}
\]

creates a 5-by-1 cell array that requires no padding because each row of the array can have a different length:
C =
   'A'
   'rolling'
   'stone'
   'gathers'
   'momentum.'

You can convert a padded character array to a cell array of strings with

\[ C = \text{cellstr}(S) \]

and reverse the process with

\[ S = \text{char}(C) \]

**Structures**

Structures are multidimensional MATLAB arrays with elements accessed by textual **field designators**. For example,

\[
\begin{align*}
S\text{.name} &= \text{'Ed Plum'}; \\
S\text{.score} &= 83; \\
S\text{.grade} &= \text{'B+'}
\end{align*}
\]

creates a scalar structure with three fields:

\[
S = 
   \begin{align*}
   \text{name} &: \text{'Ed Plum'} \\
   \text{score} &: 83 \\
   \text{grade} &: \text{'B+'}
   \end{align*}
\]

Like everything else in the MATLAB environment, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

\[
\begin{align*}
S(2)\text{.name} &= \text{'Toni Miller'}; \\
S(2)\text{.score} &= 91; \\
S(2)\text{.grade} &= \text{'A-'};
\end{align*}
\]

or an entire element can be added with a single statement:

\[
S(3) = \text{struct('name','Jerry Garcia',... 'score',70,'grade','C')}
\]
Now the structure is large enough that only a summary is printed:

```matlab
S =
1x3 struct array with fields:
    name
    score
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are mostly based on the notation of a *comma-separated list*. If you type

```matlab
S.score
```

it is the same as typing

```matlab
S(1).score, S(2).score, S(3).score
```

which is a comma-separated list.

If you enclose the expression that generates such a list within square brackets, MATLAB stores each item from the list in an array. In this example, MATLAB creates a numeric row vector containing the `score` field of each element of structure array `S`:

```matlab
scores = [S.score]
scores =
     83   91   70
```

```matlab
avg_score = sum(scores)/length(scores)
avg_score =
     81.3333
```

To create a character array from one of the text fields (`name`, for example), call the `char` function on the comma-separated list produced by `S.name`:

```matlab
names = char(S.name)
names =
    Ed Plum
    Toni Miller
    Jerry Garcia
```
Similarly, you can create a cell array from the name fields by enclosing the list-generating expression within curly braces:

```matlab
names = {S.name}
names =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

To assign the fields of each element of a structure array to separate variables outside of the structure, specify each output to the left of the equals sign, enclosing them all within square brackets:

```matlab
[N1 N2 N3] = S.name
N1 =
    Ed Plum
N2 =
    Toni Miller
N3 =
    Jerry Garcia
```

### Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```matlab
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate `expression` into a field name and obtain the values of that field at columns 1 through 25 of row 7, use

```matlab
structName.(expression)(7,1:25)
```

**Dynamic Field Names Example.** The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```matlab
function avg = avgscore(testscores, student, first, last)
    for k = first:last
        scores(k) = testscores.(student).week(k);
    end
```
end
avg = sum(scores)/(last - first + 1);

You can run this function using different values for the dynamic field student. First, initialize the structure that contains scores for a 25-week period:

testscores.Ann_Lane.week(1:25) = ... 
    [ 95 89 76 82 79 92 94 92 89 81 75 93 ...  
    85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ... 
    [ 87 80 91 84 99 87 93 87 97 87 82 89 ...  
    86 82 90 98 75 79 92 84 90 93 84 78 81];

Now run avgscore, supplying the students name fields for the testscores structure at runtime using dynamic field names:

    avgscore(testscores, 'Ann_Lane', 7, 22) 
    ans = 
          85.2500

    avgscore(testscores, 'William_King', 7, 22) 
    ans = 
          87.7500
Scripts and Functions

Overview

The MATLAB product provides a powerful programming language, as well as an interactive computational environment. You can enter commands from the language one at a time at the MATLAB command line, or you can write a series of commands to a file that you then execute as you would any MATLAB function. Use the MATLAB Editor or any other text editor to create your own function files. Call these functions as you would any other MATLAB function or command.

There are two kinds of program files:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

If you are a new MATLAB programmer, just create the program files that you want to try out in the current folder. As you develop more of your own files,
you will want to organize them into other folders and personal toolboxes that you can add to your MATLAB search path.

If you duplicate function names, MATLAB executes the one that occurs first in the search path.

To view the contents of a program file, for example, myfunction.m, use

    type myfunction

For more detailed information, see “Functions and Scripts” in the MATLAB Programming Fundamentals documentation.

**Scripts**

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like `plot`.

For example, create a file called `magicrank.m` that contains these MATLAB commands:

```matlab
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
    r(n) = rank(magic(n));
end
r
bar(r)
```

Typing the statement

    magicrank

causes MATLAB to execute the commands, compute the rank of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables `n` and `r` remain in the workspace.
Functions

Functions are files that can accept input arguments and return output arguments. The names of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is provided by `rank`. The file `rank.m` is available in the folder

```
 toolbox/matlab/matfun
```

You can see the file with

```
 type rank
```

Here is the file:

```matlab
function r = rank(A,tol)
%  RANK Matrix rank.
%  RANK(A) provides an estimate of the number of linearly
%  independent rows or columns of a matrix A.
```
% RANK(A,tol) is the number of singular values of A
% that are larger than tol.
% RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
   tol = max(size(A)) * max(s) * eps;
end
r = sum(s > tol);

The first line of a function starts with the keyword function. It gives the
function name and order of arguments. In this case, there are up to two input
arguments and one output argument.

The next several lines, up to the first blank or executable line, are comment
lines that provide the help text. These lines are printed when you type

    help rank

The first line of the help text is the H1 line, which MATLAB displays when
you use the lookfor command or request help on a folder.

The rest of the file is the executable MATLAB code defining the function. The
variable s introduced in the body of the function, as well as the variables on
the first line, r, A and tol, are all local to the function; they are separate from
any variables in the MATLAB workspace.

This example illustrates one aspect of MATLAB functions that is not
ordinarily found in other programming languages—a variable number of
arguments. The rank function can be used in several different ways:

    rank(A)
    r = rank(A)
    r = rank(A,1.e-6)

Many functions work this way. If no output argument is supplied, the result
is stored in ans. If the second input argument is not supplied, the function
computes a default value. Within the body of the function, two quantities
named nargin and nargout are available that tell you the number of input
and output arguments involved in each particular use of the function. The
rank function uses nargin, but does not need to use nargout.