CHAPTER 8

Input/Output Functions

In Chapter 2, we learned how to load and save MATLAB data using the `load` and `save` commands, and how to write out formatted data using the `fprintf` function. In this chapter we will learn more about MATLAB's input/output capabilities. First, we will learn about `textread` and `textscan`, two very useful functions for reading ASCII data from a file. Then, we will spend a bit more time examining the `load` and `save` commands. Finally, we will look at the other file I/O options available in MATLAB.

Those readers familiar with C will find much of this material very familiar. However, be careful—there are subtle differences between MATLAB and C functions that can trip you up.

8.1 The `textread` Function

The `textread` function reads ASCII files that are formatted into columns of data, where each column can be of a different type, and stores the contents of each column in a separate output array. This function is *very* useful for importing tables of data printed out by other applications.

The form of the `textread` function is

```
[a,b,c,...] = textread(filename,format,n)
```

where `filename` is the name of the file to open, `format` is a string containing a description of the type of data in each column, and `n` is the number of lines to read. (If `n` is missing, the function reads to the end of the file.) The format string contains the same types of format descriptors as function `fprintf`. Note that the number of output arguments must match the number of columns that you are reading.

For example, suppose that file `test_input.dat` contains the following data:

```
James  Jones  O+  3.51  22  Yes
Sally  Smith  A+  3.28  23  No
```

This data could be read into a series of arrays with the following function:

```
[first,last,blood,gpa,age,answer] = ...
        textread('test_input.dat','%s %s %s %f %d %s')
```

When this command is executed, the results are:

```
» [first,last,blood,gpa,age,answer] = ...
          textread('test_input.dat','%s %s %s %f %d %s')

first =
    'James'
    'Sally'
last =
    'Jones'
    'Smith'
blood =
    'O+'
    'A+'
gpa =
    3.5100
    3.2800
age =
    42
    28
answer =
    'Yes'
    'No'
```

This function can also skip selected columns by adding an asterisk to the corresponding format descriptor (for example, `%*s`). The following statement reads only the first name, last name, and gpa from the file:

```
» [first,last,gpa] = ...
          textread('test_input.dat','%s %s %*s %f %*d %*s')
first =
   'James'
   'Sally'
last =
   'Jones'
   'Smith'
gpa =
   3.5100
   3.2800
```

Function textread is much more useful and flexible than the load command. The load command assumes that all of the data in the input file is of a single type—it cannot support different types of data in different columns. In addition, it stores all of the data into a single array. In contrast, the textread function allows each column to go into a separate variable, which is *much* more convenient when working with columns of mixed data.

Function textread has a number of additional options that increase its flexibility. Consult the MATLAB on-line documentation for details of these options.

8.2 More about the load and save Commands

The save command saves MATLAB workspace data to disk, and the load command load data from disk into the workspace. The save command can save data either in a special binary format called a MAT file or in an ordinary ASCII file. The form of the save command is

```
save filename [list of variables] [options]
```

The command save all by itself saves all of the data in the current workspace to a file named matlab.mat in the current directory. If a file name is included, the data will be saved in file "filename.mat." If a list of variables is included, then only those particular variables will be saved.

For example, suppose that a workspace contains a 1000-element double array x and a character string str. We can save these two variables to a MAT file with the following command:

```
save test_matfile x str
```

This command creates a MAT file with the name test_matfile.mat. The contents of this file can be examined with -file option of the whos command:

```
» whos -file test_matfile.mat
  Name      Size      Bytes     Class
  str       1x11         22     char array
  x         1x1000     8000     double array
Grand total is 1011 elements using 8022 bytes
```

The more important options supported by the save command are shown in Table 8.1; a complete list can be found in the MATLAB on-line documentation. Note that it is possible to save disk space using a compressed MAT file with the -compress option. This is a new feature in MATLAB 7.0. It is very useful, because compressed files take up less space and can be transmitted more quickly over an network.

The load command can load data from MAT files or from ordinary ASCII files. The form of the load command is

```
save filename [options]
```

Table 8.1 Selected save Command Options

Option	Description
`-mat`	Save data in MAT-file format (default).
`-compress`	Save data in compressed format to save disk space.
`-nocompress`	Do not save data in compressed format (default).
`-ascii`	Save data in space-separated ASCII format.
`-append`	Adds the specified variables to an existing MAT file.
`-unicode`	Save character data with Unicode encoding (default).
`-nounicode`	Save character data with current system encoding.
`-v4`	Save the MAT file in a format readable by MATLAB version 4.

The command `load` all by itself loads all of the data in file `matlab.mat` into the current workspace. If a file name is included, the data will be loaded from that file name.

The options supported by the `load` command are shown in Table 8.2.

Although it is not immediately obvious, the `save` and `load` commands are the most powerful and useful I/O commands in MATLAB. Among their advantages are:

1. These commands are very easy to use.
2. MAT files are *platform independent*. A MAT file written on any type of computer that supports MATLAB can be read on any other computer supporting MATLAB. This format transfers freely among PCs, Macs, and many different versions of Linux and Unix. In addition, the Unicode character encoding ensures that character strings will be preserved properly across platforms.
3. MAT files are efficient users of disk space, using only the amount of memory required for each data type. They store the full precision of every variable—no precision is lost due to conversion two and from ASCII format. MAT files can also be compressed to save even more disk space.
4. MAT files preserve all of the information about each variable in the workspace, including its class, name, and whether or not it is global. All of this information is lost in other types of I/O. For example, suppose that the workspace contains the following information:

```
» whos
  Name        Size        Bytes    Class
  a           10x10         800    double array (global)
  ans         1x1             8    double array
  b           10x10         800    double array
  c           2x2           332    cell array
  string      1x16           32    char array
  student     1x3          2152    struct array
Grand total is 372 elements using 4124 bytes
```

Table 8.2 `load` Command Options

Option	Description
`-mat`	Treat file as a MAT file (default if file extent is `mat`).
`-ascii`	Treat file as a space-separated ASCII file (default if file extent is *not* `mat`).

If this workspace is saved with the command `save workspace.mat`, a file named `workspace.mat` will be created. When this file is loaded, all of the information will be restored, including the type of each item and whether or not it is global.

A disadvantage of these commands is that the MAT file format is unique to MATLAB and cannot be used to share data with other programs. The `-ascii` option can be used if you wish to share data with other programs, but it has serious limitations.

Good Programming Practice

Unless you must exchange data with non-MATLAB programs, always use the `load` and `save` commands to save data sets in MAT file format. This format is efficient and transportable across MATLAB implementations, and it preserves all details of all MATLAB data types.

The `save -ascii` command will not save cell or structure array data at all, and it converts string data to numbers before saving it. The `load -ascii` command will load only space-separated data with an equal number of elements on each row, and it will place all of the data into a single variable with the same name as the input file. If you need anything more elaborate (e.g., saving and loading strings, cells, structure arrays, in formats suitable for exchanging with other programs), then it will be necessary to use the other file I/O command described in this chapter.

If the file name and the names of the variables to be loaded or saved are in strings, then you should use the function forms of the `load` and `save` commands. For example, the following fragment of code asks the user for a file name and saves the workspace in that file.

```
filename = input('Enter save file name: ','s');
save (filename,'-compress');
```

8.3 An Introduction to MATLAB File Processing

To use files within a MATLAB program, we need some way to select the desired file and to read from or write to it. MATLAB has a very flexible method to read and write files, whether they are on disk, magnetic tape, or some other device attached

to the computer. This mechanism is known as the **file id** (sometimes known as **fid**). The file id is a number assigned to a file when it is opened, and used for all reading, writing, and control operations on that file. The file id is a positive integer. Two file ids are always open—file id 1 is the standard output device (`stdout`) and file id 2 is the standard error (`stderr`) device for the computer on which MATLAB is executing. Additional file ids are assigned as files are opened and released as files are closed.

Several MATLAB functions may be used to control disk file input and output. The file I/O functions are summarized in Table 8.3.

File ids are assigned to disk files or devices using the `fopen` statement, and are detached from them using the `fclose` statement. Once a file is attached to a file id using the `fopen` statement, we can read and write to that file using MATLAB file input and output statements. When we are through with the file, the `fclose` statement closes the file and makes the file id invalid. The

Table 8.3 MATLAB Input/Output Functions

Category	Function	Description
Load/Save Workspace	`load`	Load workspace.
	`save`	Save workspace.
File Opening and Closing	`fopen`	Open file.
	`fclose`	Close file.
Binary I O	`fread`	Read binary data from file.
	`fwrite`	Write binary data to file.
Formatted I/O	`fscanf`	Read formatted data from file.
	`fprintf`	Write formatted data to file.
	`fgetl`	Read line from file, discard newline character.
	`fgets`	Read line from file, keep newline character.
File Positioning, Status, and Miscellaneous	`delete`	Delete file.
	`exist`	Check for the existence of a file.
	`ferror`	Inquire file I/O error status.
	`feof`	Test for end-of-file.
	`fseek`	Set file position.
	`ftell`	Check file position.
	`frewind`	Rewind file.
Temporary Files	`tempdir`	Get temporary directory name.
	`tempname`	Get temporary file name.

frewind and fseek statements may be used to change the current reading or writing position in a file while it is open.

Data can be written to and read from files in two possible ways: as binary data or as formatted character data. Binary data consists of the actual bit patterns that are used to store the data in computer memory. Reading and writing binary data is very efficient, but a user cannot read the data stored in the file. Data in formatted files is translated into characters that can be read directly by a user. However, formatted I/O operations are slower and less efficient than binary I/O operations. We will discuss both types of I/O operations later in this chapter.

8.4 File Opening and Closing

The file opening and closing functions, fopen and fclose, are described below.

The fopen Function

The fopen function opens a file and returns a file id number for use with the file. The basic forms of this statement are

```
fid = fopen(filename,permission)
[fid, message] = fopen(filename,permission)
[fid, message] = fopen(filename,permission,format)
```

where *filename* is a string specifying the name of the file to open, *permission* is a character string specifying the mode in which the file is opened, and *format* is an optional string specifying the numeric format of the data in the file. If the open is successful, fid will contain a positive integer after this statement is executed, and message will be an empty string. If the open fails, fid will contain a −1 after this statement is executed and message will be a string explaining the error. If a file is opened for reading and it is not in the current directory, MATLAB will search for it along the MATLAB search path.

The possible permission strings are shown in Table 8.4 on the next page.

On some platforms such as PCs, it is important to distinguish between text files and binary files. If a file is to be opened in text mode, then a t should be added to the permissions string (for example, 'rt' or 'rt+'). If a file is to be opened in binary mode, a b may be added to the permissions string (for example, 'rb'), but this is not actually required, since files are opened in binary mode by default. This distinction between text and binary files does not exist on Unix or Linux computers, so the t or b is never needed on those systems.

The *format* string in the fopen function specifies the numeric format of the data stored in the file. This string is only needed when transferring files between computers with incompatible numeric data formats, so it is rarely used. A few of the possible numeric formats are shown in Table 8.5 on the next page; see the MATLAB Language Reference Manual for a complete list of possible numeric formats.

Table 8.4 fopen File Permissions

File Permission	Meaning
`'r'`	Open an existing file for reading only (default).
`'r+'`	Open an existing file for reading and writing.
`'w'`	Delete the contents of an existing file (or create a new file) and open it for writing only.
`'w+'`	Delete the contents of an existing file (or create a new file) and open it for reading and writing.
`'a'`	Open an existing file (or create a new file) and open it for writing only, appending to the end of the file.
`'a+'`	Open an existing file (or create a new file) and open it for reading and writing, appending to the end of the file.
`'W'`	Write without automatic flushing (special command for tape drives)
`'A'`	Append without automatic flushing (special command for tape drives).

There are also two forms of this function that provide information rather than open files. The function

```
fids = fopen('all')
```

returns a row vector containing a list of all file ids for currently open files (except for `stdout` and `stderr`). The number of elements in this vector is equal to the number of open files. The function

```
[filename, permission, format] = fopen(fid)
```

returns the file name, permission string, and numeric format for an open file specified by file id.

Some examples of correct `fopen` functions are shown below.

Table 8.5 fopen Format Strings

File Permission	Meaning
`'native'` or `'n'`	Numeric format for the machine MATLAB is executing on (default)
`'ieee-le'` or `'l'`	IEEE floating point with little-endian byte ordering
`'ieee-be'` or `'b'`	IEEE floating point with big-endian byte ordering
`'ieee-le.l64'` or `'a'`	IEEE floating point with little-endian byte ordering and 64-bit long data type
`'ieee-le.b64'` or `'s'`	IEEE floating point with big-endian byte ordering and 64-bit long data type

Case 1: Opening a Binary File for Input

The function below opens a file named `example.dat` for binary input only.

```
fid = fopen('example.dat','r')
```

The permission string is `'r'`, indicating that the file is to be opened for reading only. The string could have been `'rb'`, but this is not required because binary access is the default case.

Case 2: Opening a File for Text Output

The functions below open a file named `outdat` for text output only.

```
fid = fopen('outdat','wt')
```

or

```
fid = fopen('outdat','at')
```

The `'wt'` permissions string specifies that the file is a new text file; if it already exists, the old file will be deleted and a new empty file will be opened for writing. This is the proper form of the `fopen` function for an *output file* if we want to replace preexisting data.

The `'at'` permissions string specifies that we want to append to an existing text file. If it already exists, then it will be opened and new data will be appended to the currently existing information. This is the proper form of the `fopen` function for an *output file* if we don't want replace pre-existing data.

Case 3: Opening a Binary File for Read/Write Access

The function below opens a file named `junk` for binary input and output.

```
fid = fopen('junk','r+')
```

The function below also opens the file for binary input and output.

```
fid = fopen('junk','w+')
```

The difference between the first and the second statements is that the first statement required the file to exist before it is opened, while the second statement will delete any preexisting file.

Good Programming Practice

Always be careful to specify the proper permissions in `fopen` statements, depending on whether you are reading from or writing to a file. This practice will help prevent errors such as accidentally overwriting data files that you want to keep.

It is important to check for errors after you attempt to open a file. If the `fid` is −1, then the file failed to open. You should report this problem to the user, and allow him or her to either select another file or else quit the program.

Good Programming Practice

Always check the status after a file opening operation to make sure that it is successful. If the file fails to open, tell the user and provide a way to recover from the problem.

The `fclose` Function

The `fclose` function closes a file. Its form is

```
status = fclose(fid)
status = fclose('all')
```

where `fid` is a file id and `status` is the result of the operation. If the operation is successful, `status` will be 0; if it is unsuccessful, `status` will be −1.

The form `status = fclose('all')` closes all open files except for `stdout` (`fid` = 1) and `stderr` (`fid` = 2). It returns a status of 0 if all files close successfully, and −1 otherwise.

8.5 Binary I/O Functions

The binary I/O functions, `fwrite` and `fread`, are described in the following subsections.

The `fwrite` Function

The `fwrite` function writes binary data in a user-specified format to a file. Its form is

```
count = fwrite(fid,array,precision)
count = fwrite(fid,array,precision,skip)
```

where `fid` is the file id of a file opened with the `fopen` function, `array` is the array of values to write out, and `count` is the number of values written to the file.

MATLAB writes out data in *column order*, which means that the entire first column is written out, followed by the entire second column, etc. For example, if $\texttt{array} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$, the data will be written out in the order 1, 3, 5, 2, 4, 6.

The optional *`precision`* string specifies the format in which the data will be output. MATLAB supports both platform-independent precision strings, which are the same for all computers that MATLAB runs on, and platform-dependent precision strings, which vary among different types of computers. *You should use only the platform-independent strings*, and those are the only forms presented in this book.

Table 8.6 Selected MATLAB Precision Strings

MATLAB Precision String	C/Fortran Equivalent	Meaning
`'char'`	`'char*1'`	8-bit characters
`'schar'`	`'signed char'`	8-bit signed character
`'uchar'`	`'unsigned char'`	8-bit unsigned character
`'int8'`	`'integer*1'`	8-bit integer
`'int16'`	`'integer*2'`	16-bit integer
`'int32'`	`'integer*4'`	32-bit integer
`'int64'`	`'integer*8'`	64-bit integer
`'uint8'`	`'integer*1'`	8-bit unsigned integer
`'uint16'`	`'integer*2'`	16-bit unsigned integer
`'uint32'`	`'integer*4'`	32-bit unsigned integer
`'uint64'`	`'integer*8'`	64-bit unsigned integer
`'float32'`	`'real*4'`	32-bit floating point
`'float64'`	`'real*8'`	64-bit floating point
`'bitN'`		N-bit signed integer, $1 \le N \le 64$
`'ubitN'`		N-bit unsigned integer, $1 \le N \le 64$

For convenience, MATLAB accepts some C and Fortran data type equivalents for the MATLAB precision strings. If you are a C or Fortran programmer, you may find it more convenient to use the names of the data types in the language that you are most familiar with.

The possible platform-independent precisions are presented in Table 8.6. All of these precisions work in units of bytes, except for `'bitN'` or `'ubitN'`, which work in units of bits.

The optional argument *skip* specifies the number of bytes to skip in the output file before each write. This option is useful for placing values at certain points in fixed-length records. Note that if *precision* is a bit format like `'bitN'` or `'ubitN'`, skip is specified in bits instead of bytes.

The `fread` Function

The `fread` function reads binary data in a user-specified format from a file, and returns the data in a (possibly different) user-specified format. Its form is

```
[array,count] = fread(fid,size,precision)
[array,count] = fread(fid,size,precision,skip)
```

where `fid` is the file id of a file opened with the `fopen` function, `size` is the number of values to read, `array` is the array to contain the data, and `count` is the number of values read from the file.

The optional argument *`size`* specifies the amount of data to be read from the file. There are three versions of this argument:

- `n`—Read exactly `n` values. After this statement, `array` will be a column vector containing `n` values read from the file.
- `Inf`—Read until the end of the file. After this statement, `array` will be a column vector containing all of the data until the end of the file.
- `[n m]`—Read exactly `n` × `m` values, and format the data as an `n` × `m` array.

If `fread` reaches the end of the file and the input stream does not contain enough bits to write out a complete array element of the specified precision, `fread` pads the last byte or element with zero bits until the full value is obtained. If an error occurs, reading is done up to the last full value.

The *`precision`* argument specifies both the format of the data on the disk and the format of the data array to be returned to the calling program. The general form of the precision string is

```
'disk_precision => array_precision'
```

where `disk_precision` and `array_precision` are both among the precision strings found in Table 8.6. The `array_precision` value can be defaulted. If it is missing, then the data is returned in a `double` array. There is also a shortcut form of this expression if the disk precision and the array precision are the same: `'*disk_precision'`.

A few examples of `precision` strings are shown below:

`'single'`	Read data in single precision format from disk, and return it in a `double` array.
`'single=>single'`	Read data in single precision format from disk, and return it in a `single` array.
`'*single'`	Read data in single precision format from disk, and return it in a `single` array (a shorthand version of the previous string).
`'double=>real*4'`	Read data in double precision format from disk, and return it in a `single` array.

Example 8.1—Writing and Reading Binary Data

The example script file shown below creates an array containing 10,000 random values, opens a user-specified file for writing only, writes the array to disk in 64-bit floating-point format, and closes the file. It then opens the file for reading and

reads the data back into a 100 × 100 array. It illustrates the use of binary I/O operations.

```
% Script file: binary_io.m
%
% Purpose:
%   To illustrate the use of binary i/o functions.
%
% Record of revisions:
%   Date          Programmer        Description of change
%   ====          ==========        =====================
%   01/21/04    S. J. Chapman       Original code
%
% Define variables:
%   count         -- Number of values read / written
%   fid           -- File id
%   filename      -- File name
%   in_array      -- Input array
%   msg           -- Open error message
%   out_array     -- Output array
%   status        -- Operation status

% Prompt for file name
filename = input('Enter file name: ','s');

% Generate the data array
out_array = randn(1,10000);

% Open the output file for writing.
[fid,msg] = fopen(filename,'w');

% Was the open successful?
if fid > 0

    % Write the output data.
    count = fwrite(fid,out_array,'float64');

    % Tell user
    disp([int2str(count) ' values written...']);

    % Close the file
    status = fclose(fid);

else

    % Output file open failed. Display message.
    disp(msg);

end
```

```
% Now try to recover the data. Open the
% file for reading.
[fid,msg] = fopen(filename,'r');

% Was the open successful?
if fid > 0

   % Write the output data.
   [in_array, count] = fread(fid,[100 100],'float64');

   % Tell user
   disp([int2str(count) ' values read...']);

   % Close the file
   status = fclose(fid);

else

   % Input file open failed. Display message.
   disp(msg);

end
```

When this program is executed, the result are

```
» binary_io
Enter file name: testfile
10000 values written...
10000 values read...
```

An 80,000-byte file named `testfile` was created in the current directory. This file is 80,000 bytes long because it contains 10,000 64-bit values, and each value occupies 8 bytes.

◀

Quiz 8.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 8.1 through 8.5. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Why is the `textread` function especially useful for reading data created by programs written in other languages?
2. What are the advantages and disadvantages of saving data in a MAT file?
3. What MATLAB functions are used to open and close files? What is the difference between opening a binary file and opening a text file?

4. Write the MATLAB statement to open a preexisting file named `myinput.dat` for appending new text data.
5. Write the MATLAB statements required to open an unformatted input file for reading only. Check to see if the file exists and generate an appropriate error message if it doesn't.

For questions 6 and 7, determine whether the MATLAB statements are correct or not. If they are in error, specify what is wrong with them.

6.
```
fid = fopen('file1','rt');
array = fread(fid,Inf)
fclose(fid);
```

7.
```
fid = fopen('file1','w');
x = 1:10;
count = fwrite(fid,x);
fclose(fid);
fid = fopen('file1','r');
array = fread(fid,[2 Inf])
fclose(fid);
```

8.6 Formatted I/O Functions

The formatted I/O functions are described in the following subsections.

The `fprintf` Function

The `fprintf` function writes formatted data in a user-specified format to a file. Its form is

```
count = fprintf(fid,format,val1,val2,...)
fprint(format,val1,val2,...)
```

where `fid` is the file id of a file to which the data will be written, and `format` is the format string controlling the appearance of the data. If `fid` is missing, the data is written to the standard output device (the Command Window). This is the form of `fprintf` that we have been using since Chapter 2.

The format string specifies the alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters along with special sequences of characters that specify the exact format in which the output data will be displayed. The structure of a typical format is shown in Figure 8.1. A single `%` character always marks the beginning of a format—if an ordinary % sign is to be printed out, then it must appear in the format string as `%%`. After the % character, the format can have a flag, a field-width and precision specifier, and a conversion specifier. The `%` character and the conversion specifier are always required in any format, while the field, field-width, and precision specifier are optional.

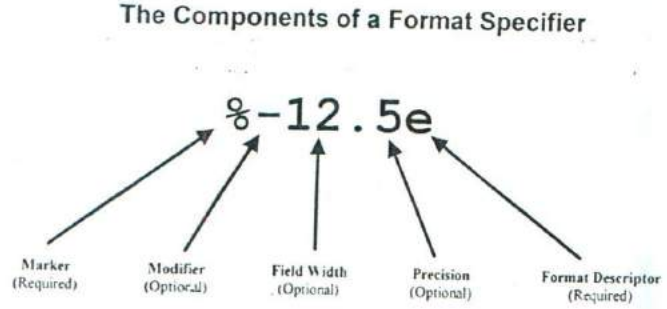


Figure 8.1 The structure of a typical format specifier.

The possible conversion specifiers are listed in Table 8.7, and the possible flags are listed in Table 8.8. If a field width and precision are specified in a format, then the number before the decimal point is the field width, which is the number of characters used to display the number. The number after the decimal point is the precision, which is the minimum number of significant digits to display after the decimal point.

In addition to ordinary characters and formats, certain special escape characters can be used in a format string. These special characters are listed in Table 8.9 on page 376.

Understanding Format Conversion Specifiers

The best way to understand the wide variety of format conversion specifiers is by example, so we will now present several examples along with their results.

Table 8.7 Format Conversion Specifiers for `fprintf`

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1416e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1416E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`. Insignificant zeros do not print.
`%G`	Same as `%g`, but using an uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a–f`)
`%X`	Hexadecimal notation (using uppercase letters `A–F`)

Table 8.8 Format Flags

Flag	Description
Minus sign (−)	Left-justifies the converted argument in its field (e.g.,: `%-5.2d`). If this flag is not present, the argument is right justified.
+	Always print a + or − sign (e.g.,: `%+5.2d`).
0	Pad argument with leading zeros instead of blanks (e.g., `%05.2d`).

Case 1: Displaying Decimal Data

Decimal (integer) data is displayed with the `%d` format conversion specifier. The d may be preceded by a flag and a field width and precision specifier, if desired. If used, the precision specifier set a minimum number of digits to display. If there are not enough digits, leading zeros will be added to the number.

Function	Result	Comment
`Fprintf('%d\n',123)`	`----|----|` `123`	Display the number using as many characters as required. For the number 123, three characters are required.
`fprintf('%6d\n',12`	`----|----|` `123`	Display the number in a 6-character-wide field. By default the number is *right justified* in the field.
`fprintf('%6.4d\n',123)`	`----|----|` `0123`	Display the number in a 6-character-wide field using a minimum of four characters. By default the number is *right justified* in the field.
`fprintf('%-6.4d\n',123)`	`----|----|` `0123`	Display the number in a 6-character-wide field using a minimum of four characters. The number is *left justified* in the field.
`fprintf('%+6.4d\n',123)`	`----|----|` `+0123`	Display the number in a 6-character-wide field using a minimum of four characters plus a sign character. By default the number is *right justified* in the field.

If a non-decimal number is displayed with the `%d` conversion specifier, the specifier will be ignored and the number will be displayed in exponential format. For example,

```
fprintf('%6d\n',123.4)
```

produces the result `1.234000e+002`.

Table 8.9 Escape Characters in Format Strings

Escape Sequences	Description
`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Print an ordinary backslash (\) symbol
`\'` or `''`	Print an apostrophe or single quote
`%%`	Print an ordinary percent (%) symbol

Case 2: Displaying Floating-Point Data

Floating-point data can be displayed with the `%e`, `%f`, or `%g` format conversion specifiers. They may be preceded by a flag and a fieldwidth and a precision specifier, if desired. If the specified field width is too small to display the number, it is ignored. Otherwise, the specified field width is used.

Function	Result	Comment
`fprintf('%f\n',123.4`	`----|----|` `123.400000`	Display the number using as many characters as required. The default case for `%f` is to display six digits after the decimal place.
`fprintf('%8.2f\n',123.4)`	`----|----|` `123.40`	Display the number in an 8-character wide field, with two places after the decimal point. The number is *right justified* in the field.
`fprintf('%4.2f\n',123.4)`	`----|----|` `123.40`	Display the number in a 6-character-wide field. The width specification was ignored because it was too small to display the number.
`fprintf('%10.2e\n',123.4)`	`----|----|` `1.23e+002`	Display the number in exponential format in a 10-character-wide field using two decimal places. By default the number is *right justified* in the field.
`fprintf('%10.2E\n',123.4)`	`----|----|` `1.23E+002`	The same but with a capital `E` for the exponent.

Case 3: Displaying Character Data

Character data may be displayed with the `%c` or `%s` format conversion specifiers. They may be preceded by a field-width specifier, if desired. If the specified field with is too small to display the number, it is ignored. Otherwise, the specified field width is used.

Function	Result	Comment
`fprintf('%c\n','s')`	`----|----|` `s`	Display a single character.
`fprintf('%s\n','string')`	`----|----|` `string`	Display the character string.
`fprintf('%8s\n','string')`	`----|----|` `  string`	Display the character string in an 8-character-wide field. By default the string is *right justified* in the field.
`fprintf('%-8s\n','string')`	`----|----|` `string`	Display the character string in an 8-character-wide field. The string is *left justified* in the field.

How Format Strings Are Used

The `fprintf` function contains a format string followed by zero or more values to print out. When the `fprintf` function is executed, the list of output values associated with the `fprintf` function is processed together with the format string. The function begins at the left end of the variable list and the left end of the format string, and scans from left to right, associating the first value in the output list with the first format descriptor in the format string, and so forth. The variables in the output list must be of the same type and in the same order as the format descriptors in the format, or unexpected results may be produced. For example, if we attempt to display a floating-point number such as 123.4 with a `%c` or `%d` descriptor, the descriptor is ignored totally and the number is printed in exponential notation.

Programming Pitfalls

Make sure that there is a one-to-one correspondence between the types of the data in an `fprintf` function and the types of the format conversion specifiers in the associated format string; otherwise, your program will produce unexpected results.

As the program moves from left to right through the variable list of an `fprintf` function, it also scans from left to right through the associated format string. Format strings are scanned according to the following rules:

1. *Format strings are scanned in order from left to right.* The first format conversion specifier in the format string is associated with the first value

in the output list of the `fprintf` function, and so forth. The type of each format conversion specifier must match the type of the data being output. In the example shown below, specifier `%d` is associated with variable `a`, `%f` with variable `b`, and `%s` with variable `c`. Note that the specifier types match the data types.

```
a = 10; b = pi; c = 'Hello';
fprintf('Output: %d %f %s\n',a,b,c);
```

2. If the scan reaches the end of the format string before the `fprintf` function runs out of values, the program starts over *at the beginning of the format string*. For example, the statements

```
a = [10 20 30 40];
fprintf('Output = %4d %4d\n',a);
```

will produce the output

```
----|----|----|----|
Output =  10  20
Output =  30  40
```

When the function reaches the end of the format string after printing `a(2)`, it starts over at the beginning of the string to print `a(3)` and `a(4)`.

3. If the `fprintf` function runs out of variables before the end of the format string, *the use of the format string stops at the first format conversion specifier without a corresponding variable, or at the end of the format string, whichever comes first*. For example, the statements

```
a = 10; b = 15; c = 20;
fprintf('Output = %4d\nOutput = %4.1f\n',a,b,c);
```

will produce the output

```
Output =  10
Output =  15.0
Output =  20
Output =
```

The use of the format string stops at `%4.1f`, which is the first unmatched format conversion specifier. On the other hand, the statements

```
voltage = 20;
fprintf('Voltage = %6.2f kV.\n',voltage);
```

will produce the output

```
Voltage =  20.00 kV.
```

since there are no unmatched format conversion specifiers, and the use of the format stops at the end of the format string.

The sprintf Function

The sprintf function is exactly like fprintf, except that it writes formatted data to a character string instead of a file. Its form is

```
string = sprint(format,val1,val2,...)
```

where fid is the file id of a file to which the data will be written and format is the format string controlling the appearance of the data. This function is very useful for creating formatted data that can be displayed within a program.

Example 8.2—Generating a Table of Information

A good way to illustrate the use of fprintf functions is to generate and print out a table of data. The example script file shown below generates the square roots, squares, and cubes of all integers between 1 and 10, and presents the data in a table with appropriate headings.

```
% Script file: table.m
%
% Purpose:
%   To create a table of square roots, squares, and
%   cubes.
%
% Record of revisions:
%     Date          Programmer          Description of change
%     ====          ==========          =====================
%   01/21/04      S. J. Chapman         Original code
%
% Define variables:
%   cube            -- Cubes
%   ii              -- Index variable
%   square          -- Squares
%   square_roots    -- Square roots
%   out             -- Output array

% Print the title of the table.
fprintf(' Table of Square Roots, Squares, and Cubes\n\n');

% Print column headings
fprintf(' Number     Square Root     Square     Cube\n');
fprintf(' ======     ===========     ======     ====\n');

% Generate the required data
ii = 1:10;
square_root = sqrt(ii);
square = ii.^2;
cube = ii.^3;
```

```
% Create the output array
out = [ii' square_root' square' cube'];

% Print the data
for ii = 1:10
   fprintf (' %2d    %11.4f    %6d    %8d\n',out(ii,:));
end
```

When this program is executed, the result is

```
» table
Table of Square Roots, Squares, and Cubes

Number      Square Root      Square      Cube
======      ===========      ======      ====
  1           1.0000           1          1
  2           1.4142           4          8
  3           1.7321           9         27
  4           2.0000          16         64
  5           2.2361          25        125
  6           2.4495          36        216
  7           2.6458          49        343
  8           2.8284          64        512
  9           3.0000          81        729
 10           3.1623         100       1000
```

The `fscanf` Function

The `fscanf` function reads formatted data in a user-specified format from a file. Its form is

```
array = fscanf(fid,format)
[array, count] = fscanf(fid,format,size)
```

where `fid` is the file id of a file from which the data will be read, `format` is the format string controlling how the data is read, and `array` is the array that receives the data. The output argument `count` returns the number of values read from the file.

The optional argument *`size`* specifies the amount of data to be read from the file. There are three versions of this argument:

- `n`—Read exactly `n` values. After this statement, `array` will be a column vector containing `n` values read from the file.
- `Inf`—Read until the end of the file. After this statement, `array` will be a column vector containing all of the data until the end of the file.

- `[n m]`—Read exactly `n` × `m` values, and format the data as an `n` × `m` array.

The format string specifies the format of the data to be read. It can contain ordinary characters along with format conversion specifiers. The `fscanf` function compares the data in the file with the format conversion specifiers in the format string. As long as the two match, `fscanf` converts the value and stores it in the output array. This process continues until the end of the file or until the amount of data in *`size`* has been read, whichever comes first.

If the data in the file does not match the format conversion specifiers, the operation of `fscanf` stops immediately.

The format conversion specifiers for `fscanf` are basically the same as those for `fprintf`. The most common specifiers are shown in Table 8.10.

To illustrate the use of `fscanf`, we will attempt to read a file called `x.dat` containing the following values on two lines:

```
10.00  20.00
30.00  40.00
```

1. If the file is read with the statement

   ```
   [z, count] = fscanf(fid,'%f');
   ```

   variable `z` will be the column vector $\begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \end{bmatrix}$ and `count` will be 4.

2. If the file is read with the statement

   ```
   [z, count] = fscanf(fid,'%f',[2 2]);
   ```

   variable `z` will be the array $\begin{bmatrix} 10 & 30 \\ 20 & 40 \end{bmatrix}$ and `count` will be 4.

Table 8.10 Format Conversion Specifiers for `fscanf`

Specifier	Description
`%c`	Read a single character. This specifier reads any character including blanks, new lines, and the like.
`%Nc`	Read *N* characters.
`%d`	Read a decimal number (ignores blanks).
`%e %f %g`	Read a floating-point number (ignores blanks).
`%i`	Read a signed integer (ignores blanks).
`%s`	Read a string of characters. The string is terminated by blanks or other special characters, such as new lines.

3. Next, let's try to read this file as decimal values. If the file is read with the statement

```
[z, count] = fscanf(fid,'%d',Inf);
```

variable z will be the single value 10 and count will be 1. This happens because the decimal point in the 10.00 does not match the format conversion specifier, and fscanf stops at the first mismatch.

4. If the file is read with the statement

```
[z, count] = fscanf(fid,'%d.%d',[1 Inf]);
```

variable z will be the row vector [10 0 20 0 30 0 40 0] and count will be 8. This happens because the decimal point is now matched in the format conversion specifier and the numbers on either side of the decimal point are interpreted as separate integers!

5. Now let's try to read the file as individual characters. If the file is read with the statement

```
[z, count] = fscanf(fid,'%c');
```

variable z will be a row vector containing every character in the file, including all spaces and newline characters! Variable count will be equal to the number of characters in the file.

6. Finally, let's try to read the file as a character string. If the file is read with the statement

```
[z, count] = fscanf(fid,'%s');
```

variable z will be a row vector containing the 20 characters 10.0020.0030.0040.00, and count will be 4. This happens because the string specifier ignores white space, and the function found four separate strings in the file.

The fgetl Function

The fgetl function reads the next line *excluding the end-of-line characters* from a file as a character string. It form is

```
line = fgetl(fid)
```

where fid is the file id of a file from which the data will be read, and line is the character array that receives the data. If fgetl encounters the end of a file, the value of line is set to −1.

The `fgets` Function

The `fgets` function reads the next line *including the end-of-line characters* from a file as a character string. It form is

```
line = fgets(fid)
```

where `fid` is the file id of a file from which the data will be read and `line` is the character array that receives the data. If `fgets` encounters the end of a file, the value of `line` is set to -1.

8.7 Comparing Formatted and Binary I/O Functions

Formatted I/O operations produce formatted files. A **formatted file** contains recognizable characters, numbers, etc. stored as ASCII text. These files are easy to distinguish, because we can see the characters and numbers in the file when we display them on the screen or print them on a printer. However, to use data in a formatted file, a MATLAB program must translate the characters in the file into the internal data format used by the computer. Format conversion specifiers provide the instructions for this translation.

Formatted files have the advantages that we can readily see what sort of data they contain, and it is easy to exchange data between different types of programs using them. However, they also have disadvantages. A program must do a good deal of work to convert a number between the computer's internal representation and the characters contained in the file. All of this work is just wasted effort if we are going to be reading the data back into another MATLAB program. Also, the internal representation of a number usually requires much less space than the corresponding representation of the number found in a formatted file. For example, the internal representation of a 64-bit floating-point value requires 8 bytes of space. The character representation of the same value would be ±d.ddddddddddddddE±ee, which requires 21 bytes of space (one byte per character). Thus, storing data in character format is inefficient and wasteful of disk space.

Unformatted files (or **binary files**) overcome these disadvantages by copying the information from the computer's memory directly to the disk file with no conversions at all. Since no conversions occur, no computer time is wasted formatting the data. In MATLAB, binary I/O operations are *much* faster than formatted I/O operations because there is no conversion. Furthermore, the data occupies a much smaller amount of disk space. On the other hand, unformatted data cannot be examined and interpreted directly by humans. In addition, it usually cannot be moved between different types of computer, because computers typically have different internal ways to represent integers and floating-point values.

Formatted and unformatted files are compared in Table 8.11. In general, formatted files are best for data that people must examine, or data that may have to be moved between different programs on different computers. Unformatted files

Table 8.11 Comparison of Formatted and Unformatted Files

Formatted Files	Unformatted Files
Can display data on output devices.	Cannot display data on output devices.
Can easily transport data between different computers.	Cannot easily transport data between computers with different internal data representations.
Require a relatively large amount of disk space.	Requires relatively little disk space.
Are slow,: requiring a lot of computer time.	Are fast, requiring little computer time.
Truncation or rounding errors are possible in formatting.	No truncation or rounding errors occur.

are best for storing information that will not need to be examined by human beings, and that will be created and used on the same type of computer. Under those circumstances, unformatted files are faster and occupy less disk space.

Good Programming Practice

Use formatted files to create data that must be readable by humans or that must be transferable between programs on computers of different types. Use unformatted files to efficiently store large quantities of data that do not have to be directly examined and that will remain on only one type of computer. Also, use unformatted files when I/O speed is critical.

Example 8.3—Comparing Formatted and Binary I/O

The program shown below compares the time required to read and write a 10,000-element array using both formatted and binary I/O operations. Note that each operation is repeated ten times and the average time is reported.

```
% Script file: compare.m
%
% Purpose:
%   To compare binary and formatted I/O operations.
%   This program generates an array of 10,000 random
%   values and writes it to disk both as a binary and
%   as a formatted file.
%
```

```
% Record of revisions:
%      Date          Programmer          Description of change
%      ====          ==========          =====================
%    01/15/04      S. J. Chapman       Original code
%
% Define variables:
%   count          -- Number of values read / written
%   fid            -- File id
%   in_array       -- Input array
%   msg            -- Open error message
%   out_array      -- Output array
%   status         -- Operation status
%   time           -- Elapsed time in seconds

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate the data array.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
out_array = randn(1,10000);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% First, time the binary output operation.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Reset timer
tic;

% Loop for 10 times
for ii = 1:10

   % Open the binary output file for writing.
   [fid,msg] = fopen('unformatted.dat','w');

   % Write the data
   count = fwrite(fid,out_array,'float64');

   % Close the file
   status = fclose(fid);

end

% Get the average time
time = toc / 10;
fprintf ('Write time for unformatted file = %6.3f\n',time);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Next, time the formatted output operation.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Reset timer
tic;

% Loop for 10 times
for ii = 1:10

   % Open the formatted output file for writing.
   [fid,msg] = fopen('formatted.dat','wt');

   % Write the data
   count = fprintf(fid,'%23.15e\n',out_array);

   % Close the file
   status = fclose(fid);

end

% Get the average time
time = toc / 10;
fprintf ('Write time for formatted file = %6.3f\n',time);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Time the binary input operation.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Reset timer
tic;

% Loop for 10 times
for ii = 1:10

   % Open the binary file for reading.
   [fid,msg] = fopen('unformatted.dat','r');

   % Read the data
   [in_array, count] = fread(fid,Inf,'float64');

   % Close the file
   status = fclose(fid);

end

% Get the average time
time = toc / 10;
fprintf ('Read time for unformatted file = %6.3f\n', time);
```

```
%%%
```

Note that the write time for the formatted file was 12 times slower than the write time for the unformatted file, and the read time for the formatted file was almost 30 times slower than the read time for the unformatted file. Furthermore, the formatted file was three times larger than the unformatted file.

It is clear from these results that unless you *really* need formatted data, binary I/O operations are the preferred way to save data in MATLAB. ◀

Quiz 8.2

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 8.6 and 8.7. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the difference between unformatted (binary) and formatted I/O operations?
2. When should formatted I/O be used? When should unformatted I/O be used?
3. Write the MATLAB statements required to create a table that contains the sine and cosine of x for $x = 0, 0.1\pi, \ldots, \pi$. Be sure to include a title and label on the table.

For questions 4 and 5, determine whether the MATLAB statements are correct or not. If they are in error, specify what is wrong with them.

4.
```
a = 2*pi;
b = 6;
c = 'hello';
fprintf(fid,'%s %d %g\n',a,b,c);
```

5.
```
data1 = 1:20;
data2 = 1:20;
fid = fopen('xxx','w+');
fwrite(fid,data1);
fprintf(fid,'%g\n',data2);
```

8.8 File Positioning and Status Functions

As we stated previously, MATLAB files are sequential—they are read in order from the first record in the file to the last record in the file. However, we sometimes need to read a piece of data more than once, or to process a whole file

more than once during a program. How can we skip around within a sequential file?

The MATLAB function `exist` can determine whether or not a file exists before it is opened. There are two functions to tell us where we are within a file once it is opened: `feof` and `ftell`. In addition, there are two functions to help us move around within the file: `frewind` and `fseek`.

Finally, MATLAB includes a function `ferror` that provides a detailed description of cause of I/O errors when they occur. We will now explore these five functions, looking at `ferror` first, since it can be used with all of the other functions.

The `exist` Function

The MATLAB function `exist` checks for the existence of a variable in a workspace, a built-in function, or a file in the MATLAB search path. The forms of the `ferror` function are:

```
ident = exist('item');
ident = exist('item','kind');
```

If `'item'` exists, this function returns a value based on its type. The possible results are shown in Table 8.12.

The second form of the `exist` restricts the search for an item to a specified kind. The legal types are `'var'`, `'file'`, `'builtin'`, and `'dir'`.

The `exist` function is very important, because we can use it to check for the existence of a file before it is overwritten by `fopen`. The permissions `'w'` and `'w+'` delete the contents an existing file when they open it. Before a programmer allows `fopen` to delete an existing file, he or she should check with the user to confirm that the file really should be deleted.

Table 8.12 Values Returned by the `exist` Function

Value	Meaning
0	Item not found.
1	Item is a variable in the current workspace.
2	Item is an M-file or a file of unknown type.
3	Item is a MEX file.
4	Item is a MDL file.
5	Item is a built-in function.
6	Item is a P file.
7	Item is a directory.
8	Item is a Java class.

Example 8.4—Opening an Output File

The program shown gets an output file name from the user and checks to see whether it exists. If it exists, the program checks to see whether the user wants to delete the existing file or to append the new data to it. If the file does not exist, then the program simply opens the output file.

```
% Script file: output.m
%
% Purpose:
%   To demonstrate opening an output file properly.
%   This program checks for the existence of an output
%   file. If it exists, the program checks to see if
%   the old file should be deleted, or if the new data
%   should be appended to the old file.
%
% Record of revisions:
%     Date          Programmer        Description of change
%     ====          ==========        =====================
%   01/21/14      S. J. Chapman       Original code
%
% Define variables:
%   fid             -- File id
%   out_filename    -- Output file name
%   yn              -- Yes/No response

% Get the output file name.
out_filename = input('Enter output filename: ','s');

% Check to see if the file exists.
if exist(out_filename,'file')

   % The file exists
   disp('Output file already exists.');
   yn = input('Keep existing file? (y/n) ','s');

   if yn == 'n'
      fid = fopen(out_filename,'wt');
   else
      fid = fopen(out_filename,'at');
   end

else
```

```
    % File doesn't exist
    fid = fopen(out_filename,'wt');

end

% Output data
fprintf(fid,'%s\n',date);

% Close file
fclose(fid);
```

When this program is executed, the results are:

```
» output
Enter output filename: xxx          (Nonexistent file)
» type xxx

21-Jan-2004

» output
Enter output filename: xxx
Output file already exists.
Keep existing file? (y/n) y         (Keep current file)
» type xxx

21-Jan-2004
21-Jan-2004                         (Note new data added)

» output
Enter output filename: xxx
Output file already exists.
Keep existing file? (y/n) n         (Replace current file)
» type xxx

21-Jan-2004
```

The program appears to be functioning correctly in all three cases. ◀

Good Programming Practice

Do not overwrite an output file without confirming that the user would like to delete the preexisting information.

The ferror Function

The MATLAB I/O system has several internal variables, including a special error indicator that is associated with each open file. This error indicator is updated by every I/O operation. The `ferror` function gets the error indicator and translates it into an easy-to-understand character message. The forms of the `ferror` function are:

```
message = ferror(fid)
message = ferror(fid,'clear')
[message,errnum] = ferror(fid)
```

This function returns the most recent error message (and optionally error number) associated with the file attached to `fid`. It may be called at any time after any I/O operation to get a more detailed description of what went wrong. If this function is called after a successful operation, the message will be `''` and the error number will be 0.

The `'clear'` argument clears the error indicator for a particular file id.

The feof Function

The `feof` function tests to see if the current file position is at the end of the file. The form of the `feof` function is:

```
eofstat = feof(fid)
```

This function returns a logical true (1) if the current file position is at the end of the file, and logical false (0) otherwise.

The ftell Function

The `ftell` function returns the current location of the file position indicator for the file specified by `fid`. The position is a nonnegative integer specified in bytes from the beginning of the file. A returned value of -1 for position indicates that the query was unsuccessful. If this happens, use `ferror` to determine why the request failed. The form of the `ftell` function is:

```
position = ftell(fid)
```

The frewind Function

The `frewind` function allows a programmer to reset a file's position indicator to the beginning of the file. The form of the `frewind` function is:

```
frewind(fid)
```

This function does not return status information.

The fseek Function

The fseek function allows a programmer to set a file's position indicator to an arbitrary location within a file. The form of the fseek function is:

```
status = fseek(fid,offset,origin)
```

This function repositions the file position indicator in the file with the given fid to the byte with the specified offset relative to origin. The offset is measured in bytes, with a positive number indicating motion towards the end of the file and a negative number indicating motion towards the head of the file. The origin is a string that can have one of three possible values.

- 'bof'—This is the beginning of the file.
- 'cof'—This is the current position within the file.
- 'eof'—This is the end of the file.

The returned status is zero if the operation is successful and −1 if the operation fails. If the returned status is −1, use ferror to determine why the request failed.

As an example of using fseek and ferror together, consider the following statements.

```
[fid,msg] = fopen('x','r');
status = fseek(fid,-10,'bof');
if status ~= 0
   msg = ferror(fid);
   disp(msg);
end
```

These commands open a file and attempt to set the file pointer 10 bytes before the beginning of the file. Since this is impossible, fseek returns a status of −1, and ferror gets an appropriate error message. When these statements are executed, the result is an informative error message:

```
Offset is bad - before beginning-of-file.
```

Example 8.5—Fitting a Line to a Set of Noisy Measurements

In Example 4.6, we learned how to perform a fit of a noisy set of measurements (x, y) to a line of the form

$$y = mx + b \tag{8-1}$$

The standard method for finding the regression coefficients m and b is the method of least squares. This method is named "least squares" because it produces the line $y = mx + b$ for which the sum of the squares of the differences between the

observed y values and the predicted y values is as small as possible. The slope of the least squares line is given by

$$m = \frac{(\sum xy) - (\sum x)\bar{y}}{(\sum x^2) - (\sum x)\bar{x}} \tag{8-2}$$

and the intercept of the least squares line is given by

$$b = \bar{y} - m\bar{x} \tag{8-3}$$

where

$\sum x$ is the sum of the x values
$\sum xx^2$ is the sum of the squares of the x values
$\sum xy$ is the sum of the products of the corresponding x and y values
$\bar{x}$ is the mean (average) of the x values
$\bar{y}$ is the mean (average) of the y values

Write a program that will calculate the least-squares slope m and y-axis intercept b for a given set of noisy measured data points (x, y) *that are to be found in an input data file.*

SOLUTION

1. **State the problem.**
   Calculate the slope m and intercept b of a least-squares line that best fits an input data set consisting of an arbitrary number of (x, y) pairs. The input (x, y) data resides in a user-specified input file.

2. **Define the inputs and outputs.**
   The inputs required by this program are pairs of points (x, y), where x and y are real quantities. Each pair of points will be located on a separate line in the input disk file. The number of points in the disk file is not known in advance.

   The outputs from this program are the slope and intercept of the least-squares fitted line, plus the number of points going into the fit.

3. **Describe the algorithm.**
   This program can be broken down into four major steps

```
Get the name of the input file and open it
Accumulate the input statistics
Calculate the slope and intercept
Write out the slope and intercept
```

   The first major step of the program is to get the name of the input file and to open the file. To do this, we will have to prompt the user to enter the name of the input file. After the file is opened, we must check to see that the open was successful. Next, we must read the file and keep track of the

number of values entered, plus the sums $\sum x$, $\sum y$, $\sum x^2$, and $\sum xy$. The pseudocode for these steps is:

```
Initialize n, sum_x, sum_x2, sum_y, and sum_xy to 0
Prompt user for input file name
Open file 'filename'
Check for error on open
if no error
   READ x, y from file 'filename'
   while not at end-of-file
      n <- n + 1
      sum_x <- sum_x + x
      sum_y <- sum_y + y
      sum_x2 <- sum_x2 + x^2
      sum_xy <- sum_xy + x*y
      READ x, y from file 'filename'
   end
  (further processing)
end
```

Next, we must calculate the slope and intercept of the least-squares line. The pseudocode for this step is just the MATLAB versions of Equations (8-2) and (8-3).

```
x_bar <- sum_x / n
y_bar <- sum_y / n
slope <- (sum_xy - sum_x*y_bar) / (sum_x2 - sum_x*x_bar)
y_int <- y_bar - slope * x_bar
```

Finally, we must write out the results.

```
Write out slope 'slope' and intercept 'y_int'.
```

4. **Turn the algorithm into MATLAB statements.**
The final MATLAB program is shown below.

```
% Script file: lsqfit.m
%
% Purpose:
%   To perform a least-squares fit of an input data set
%   to a straight line, and print out the resulting slope
%   and intercept values. The input data for this fit
%   comes from a user-specified input data file.
%
% Record of revisions:
%     Date        Programmer        Description of change
%     ====        ==========        =====================
%   01/22/04    S. J. Chapman     Original code
```

```
%
% Define variables:
%   count          -- number of values read
%   filename       -- Input file name
%   fid            -- File id
%   msg            -- Open error message
%   n              -- Number of input data pairs (x,y)
%   slope          -- Slope of the line
%   sum_x          -- Sum of all input X values
%   sum_x2         -- Sum of all input X values squared
%   sum_xy         -- Sum of all input X*Y values
%   sum_y          -- Sum of all input Y values
%   x              -- An input X value
%   x_bar          -- Average X value
%   y              -- An input Y value
%   y_bar          -- Average Y value
%   y_int          -- Y-axis intercept of the line

% Initialize sums
n = 0; sum_x = 0; sum_y = 0; sum_x2 = 0; sum_xy = 0;

% Prompt user and get the name of the input file.
disp('This program performs a least-squares fit of an');
disp('input data set to a straight line. Enter the name');
disp('of the file containing the input (x,y) pairs: ' );
filename = input(' ','s');

% Open the input file
[fid,msg] = fopen(filename,'rt');

% Check to see if the open failed.
if fid < 0

   % There was an error--tell user.
   disp(msg);

else

   % File opened successfully. Read the (x,y) pairs from
   % the input file. Get first (x,y) pair before the
   % loop starts.
   [in,count] = fscanf(fid,'%g %g',2);

   while ~feof(fid)
      x = in(1);
      y = in(2);
      n     = n + 1;                    %
      sum_x = sum_x + x;                % Calculate
      sum_y = sum_y + y;                % statistics
```

```
        sum_x2 = sum_x2 + x.^2;          %
        sum_xy = sum_xy + x * y;         %

        % Get next (x,y) pair
        [in,count] = fscanf(fid,'%f',[1 2]);

    end

    % Close the file
    fclose(fid);

    % Now calculate the slope and intercept.
    x_bar = sum_x / n;
    y_bar = sum_y / n;
    slope = (sum_xy - sum_x*y_bar) / (sum_x2 - sum_x*x_bar);
    y_int = y_bar - slope * x_bar;

    % Tell user.
    fprintf('Regression coefficients for the least-squares line:\n');
    fprintf(' Slope (m)      = %12.3f\n',slope);
    fprintf(' Intercept (b) = %12.3f\n',y_int);
    fprintf(' No of points   = %12d\n',n);
end
```

5. **Test the program.**
To test this program, we will try a simple data set. For example, if every point in the input data set actually falls along a line, then the resulting slope and intercept should be exactly the slope and intercept of that line. Thus the data set

```
1.1  1.1
2.2  2.2
3.3  3.3
4.4  4.4
5.5  5.5
6.6  6.6
7.7  7.7
```

should produce a slope of 1.0 and an intercept of 0.0. If we place these values in a file called `input1`, and run the program, the results are:

```
» lsqfit
This program performs a least-squares fit of an
input data set to a straight line. Enter the name
of the file containing the input (x,y) pairs:
input1
Regression coefficients for the least-squares line:
   Slope (m)      = 1.000
   Intercept (b) = 0.000
   No of points   =     7
```

Now let's add some noise to the measurements. The data set becomes

```
1.1  1.01
2.2  2.30
3.3  3.05
4.4  4.28
5.5  5.75
6.6  6.48
7.7  7.84
```

If these values are placed in a file called `input2`, and the program is run on that file, the results are:

```
» lsqfit
This program performs a least-squares fit of an
input data set to a straight line. Enter the name
of the file containing the input (x,y) pairs:
input2
Regression coefficients for the least-squares line:
   Slope (m)      =  1.024
   Intercept (b)  = -0.120
   No of points   =      7
```

If we calculate the answer by hand, it is easy to show that the program gives the correct answers for our two test data sets. The noisy input data set and the resulting least-squares fitted line are shown in Figure 8.2.

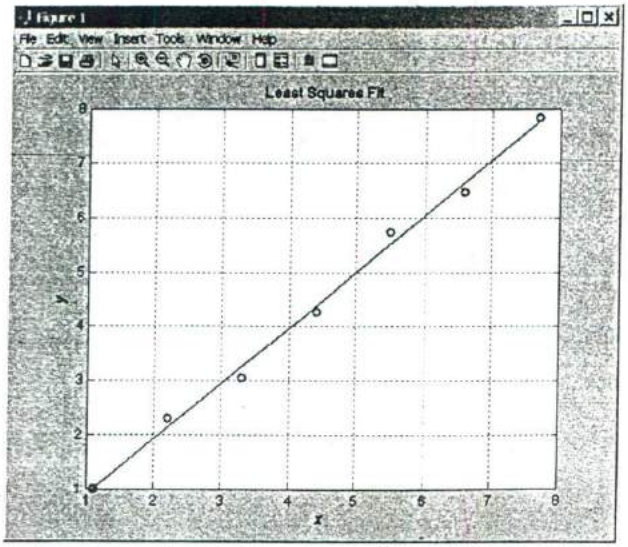


Figure 8.2 A noisy input data set and the resulting least-squares fitted line.

The textscan Function

The textscan function reads ASCII files that are formatted into columns of data, where each column can be of a different type, and stores the contents into the columns of a cell array. This function is *very* useful for importing tables of data printed out by other applications. It is new in MATLAB 7.0. It is basically similar to textread, except that it is faster and more flexible.

The form of the textscan function is

```
a = textscan(fid, 'format')
a = textscan(fid, 'format', N)
a = textscan(fid, 'format', param, value, ...)
a = textscan(fid, 'format', N, param, value, ...)
```

where fid is the file id of a file that has already been opened with fopen, format is a string containing a description of the type of data in each column, and n is the number of times to use the format specifier. (If n is −1 or is missing, the function reads to the end of the file.) The format string contains the same types of format descriptors as function fprintf. Note that there is only one output argument, with all of the values returned in a cell array. The cell array will contain a number of elements equal to the number of format descriptors to read.

For example, suppose that file test_input1.dat contains the following data:

```
James     Jones     O+    3.51    22    Yes
Sally     Smith     A+    3.28    23    No
Hans      Carter    B-    2.84    19    Yes
Sam       Spade     A+    3.12    21    Yes
```

This data could be read into a cell array with the following function:

```
fid = fopen('test_input1.dat','rt');
a = textscan(fid,'%s %s %s %f %d %s',-1);
fclose(fid);
```

When this command is executed, the results are:

```
» fid = fopen('test_input1.dat','rt');
» a = textscan(fid,'%s %s %s %f %d %s',-1)

a =
  {4x1 cell}  {4x1 cell}  {4x1 cell}  [4x1 double]
  [4x1 int32]  {4x1 cell}
» a{1}
ans =
    'James'
    'Sally'
    'Hans'
    'Sam'
```

```
» a{2}
ans =
    'Jones'
    'Smith'
    'Carter'
    'Spade'
» a{3}
ans =
    'O+'
    'A+'
    'B-'
    'A+'
» a{4}
ans =
    3.5100
    3.2800
    2.8400
    3.1200
» fclose(fid);
```

This function can also skip selected columns by adding an asterisk to the corresponding format descriptor (for example, `%*s`). For example, the following statements read only the first name, last name, and gpa from the file:

```
fid = fopen('test_input1.dat','rt');
a = textscan(fid,'%s %s %*s %f %*d %*s',-1);
fclose(fid);
```

Function `textscan` is similar to function `textread`, but it is more flexible and faster. The advantages of `textscan` include:

1. The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.
2. With `textscan`, you can start reading at any point in the file. When the file is opened with `fopen`, you can move to any position in the file with `fseek` and begin the `textscan` at that point. The `textread` function requires that you start reading from the beginning of the file.
3. Subsequent `textscan` operations start reading the file at a point where the last `textscan` left off. The `textread` function always begins at the start of the file, regardless of any prior `textread` operations.
4. Function `textscan` returns a single cell array regardless of how many fields you read. With `textscan`, you don't need to match the number of output arguments with the number of fields being read, as you would with `textread`.
5. Function `textscan` offers more choices in how the data being read is converted.

Function `textscan` has a number of additional options that increase its flexibility. Consult the MATLAB on-line documentation for details of these options.

Good Programming Practice

Use function `textscan` in preference to `textread` to import ASCII data in column format from programs written in other languages or exported from applications such as spreadsheets.

Function `uiimport`

Function `uiimport` is a GUI-based way to import data from a file or from the clipboard. This command takes the forms

```
uiimport
structure = uiimport;
```

In the first case, the imported data is inserted directly into the current MATLAB workspace. In the second case, the data is converted into a structure and saved in variable `structure`.

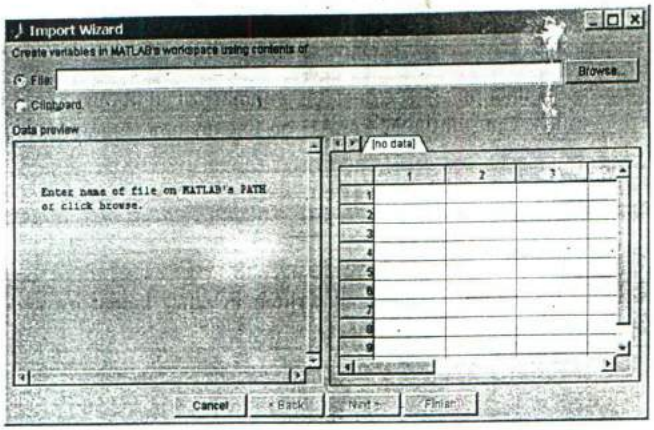
When the command `uiimport` is typed, the Import Wizard is displayed in a window (see Figure 8.3 for the PC version of this window on the next page). The user can then select the file that he or she would like to import from and the specific data within that file. Many different formats are supported—a partial list is given in Table 8.13 on page 403. In addition, data can be imported from almost *any* application by saving the data on the clipboard. This flexibility can be very useful when you are trying to get data into MATLAB for analysis.

8.9 Summary

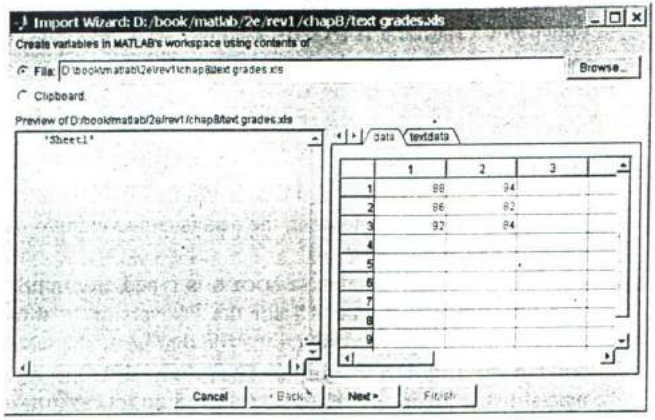
In Chapter 8, we presented an introduction to file I/O operations. Many MATLAB I/O functions are quite similar to C functions, but there are differences in some details.

The `textread` and `textscan` functions can be used to import ASCII data in column format from programs written in other languages or exported from applications such as spreadsheets. Of these two functions, `textscan` is preferred because it is more flexible and faster than `textread`.

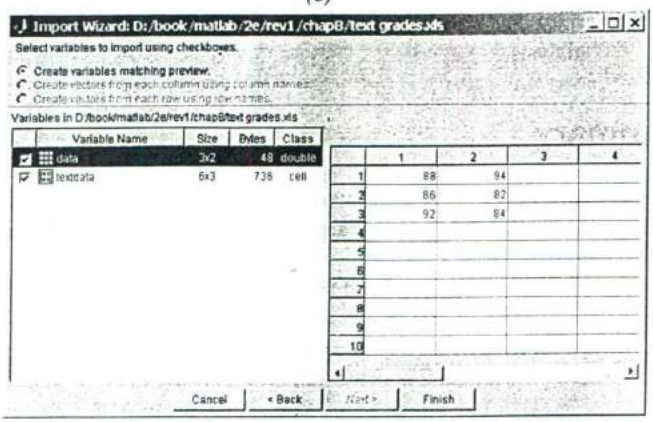
The `load` and `save` commands using MAT files are very efficient, are transportable across MATLAB implementations, and preserve full precision, data types, and global status for all variables. MAT files should be used as the first-choice method of I/O, unless data must be shared with other applications or must be readable by humans.



(a)



(b)



(c)

Figure 8.3 Using uiimport: *(a)* The Import Wizard after it had been started. *(b)* After a data file has been selected, one or more data arrays are created, and their contents can be examined. *(c)* Next, the user can select which of the data arrays will be imported into MATLAB.

There are two types of I/O statements in MATLAB: binary and formatted. Binary I/O statements store or read data in unformatted files, and formatted I/O statements store or read data in formatted files.

MATLAB files are opened with the `fopen` function and closed with the `fclose` function. Binary reads and writes are performed with the `fread` and `fwrite` functions, while formatted reads and writes are performed with the `fscanf` and `fprintf` functions. Functions `fgets` and `fgetl` simply transfer a line of text from a formatted file into a character string.

The `exist` function can be used to determine whether a file exists before it is opened. This is useful to ensure that existing data is not accidentally overwritten.

It is possible to move around within a disk file using the `frewind` and `fseek` functions. The `frewind` function moves the current file position to the beginning of the file, while the `fseek` function move the current file position to

Table 8.13 Selected File Formats Supported by `uiimport`

File Extents	Meaning
`*.gif`	Image files
`*.jpg`	
`*.jpeg`	
`*.ico`	
`*.png`	
`*.pcx`	
`*.tif`	
`*.tiff`	
`*.bmp`	
`*.cur`	Cursor format
`*.hdf`	Hierarchical Data Format file
`*.au`	Sound files
`*.snd`	
`*.wav`	
`*.avi`	Movie file
`*.csv`	Spreadsheet files
`*.xls`	
`*.wk1`	
`*.txt`	Text files
`*.dat`	
`*.dlm`	
`*.tab`	

a point a specified number of bytes ahead or behind a reference point. The reference point may be the current file position, the beginning of the file, or the end of the file.

Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB I/O functions.

1. Unless you must exchange data with non-MATLAB programs, always use the `load` and `save` commands to save data sets in MAT file format. This format is efficient and transportable across MATLAB implementations, and it preserves all details of all MATLAB data types.
2. Always be careful to specify the proper permissions in `fopen` statements, depending on whether you are reading from or writing to a file. This practice will help prevent errors such as accidentally overwriting data files that you want to keep.
3. Always check the status after a file-opening operation to make sure that it is successful. If the file opening fails, tell the user and provide a way to recover from the problem.
4. Use formatted files to create data that must be readable by humans or that must be transferable between programs on computers of different types. Use unformatted files to efficiently store large quantities of data that do not have to be directly examined and that will remain on only one type of computer. Also, use unformatted files when I/O speed is critical.
5. Do not overwrite an output file without confirming that the user would like to delete the preexisting information.
6. Use function `textscan` in preference to `textread` to import ASCII data in column format from programs written in other languages or exported from applications such as spreadsheets.

MATLAB Summary

The summary at the top of the next page lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

8.10 Exercises

8.1 What is the difference between binary and formatted I/O? Which MATLAB functions perform each type of I/O?

8.2 **Table of Logarithms** Write a MATLAB program to generate a table of the base-10 logarithms between 1 and 10 in steps of 0.1. The table should start in a new page, and it should include a title describing the table and row and column headings. This table should be organized as shown in the table in the middle of the next page.

`exist`	Check for the existence of a file.
`fclose`	Close file.
`feof`	Test for end-of-file.
`ferror`	Inquire file I/O error status.
`fgetl`	Read a line from file, and discard newline character.
`fgets`	Read a line from file, and keep newline character.
`fopen`	Open file.
`fprintf`	Write formatted data to file.
`fread`	Read binary data from file.
`frewind`	Rewind file.
`fscanf`	Read formatted data from file.
`fseek`	Set file position.
`ftell`	Check file position.
`fwrite`	Write binary data to a file.
`sprintf`	Write formatted data to a character string.
`textread`	Read data of various types organized in column format from an ASCII file, and store the data in each column in separate variables.
`textscan`	Read data of various types organized in column format from an ASCII file, and store the data in a cell array.

	X.0	X.1	X.2	X.3	X.4	X.5	X.6	X.7	X.8	X.9
1.0	0.000	0.041	0.079	0.114	...					
2.0	0.301	0.322	0.342	0.362	...					
3.0	...									
4.0	...									
5.0	...									
6.0	...									
7.0	...									
8.0	...									
9.0	...									
10.0	...									

8.3 Write a MATLAB program that reads in a time in seconds since the start of the day (this value will be somewhere between 0. and 86,400.), and prints a character string containing time in the form `HH:MM:SS` using the 24-hour clock convention. Use the proper format converter to ensure that leading zeros are preserved in the `MM` and `SS` fields. Also, be sure to check the input number of seconds for validity, and write an appropriate error message if an invalid number is entered.

8.4 **Gravitational Acceleration** The acceleration due to the Earth's gravity at any height h above the surface of the Earth is given by the equation

$$g = -G\frac{M}{(R + h)^2} \tag{8-4}$$

where G is the gravitational constant (6.672×10^{-11} N m^2/kg^2), M is the mass of the earth (5.98×10^{24} kg), R is the mean radius of the Earth (6371 km), and h is the height above the Earth's surface. If M is measured in kg and R and h in meters, then the resulting acceleration will be in units of meters per second squared. Write a program to calculate the acceleration due to the Earth's gravity in 500 km increments at heights from 0 km to 40,000 km above the surface of the Earth. Print out the results in a table of height versus acceleration with appropriate labels, including the units of the output values. Plot the data as well.

8.5 The program in Example 8.5 illustrated the use of formatted I/O commands to read (x, y) pairs of data from disk. However, this could also be done with the `load -ascii` function. Rewrite this program to use `load` instead of the formatted I/O functions. Test your rewritten program to confirm that it gives the same answers as Example 8.5.

8.6 Rewrite the program in Example 8.5 to use the `textread` function instead of the formatted I/O functions.

8.7 Rewrite the program in Example 8.5 to use the `textscan` function instead of the formatted I/O functions. How difficult was it to use `textscan`, compared with using `textread`, `load -ascii`, or the formatted I/O functions?

8.8 Write a program that reads an arbitrary number of real values from a user-specified input data file, rounds the values to the nearest integer, and writes the integers out to a user-specified output file. Make sure that the input file exists, and if not, tell the user and ask for another input file. If the output file exists, ask the user whether or not to delete it. If not, prompt for a different output file name.

8.9 **Table of Sines and Cosines** Write a program to generate a table containing the sine and cosine of θ for θ between 0° and 90°, in 1° increments. The program should properly label each of the columns in the table.

8.10 File `int.dat` (available at the book's Web site) contains 25 integer values in `'int8'` format. Write a program that reads these values into a `single` array using function `fread`.

8.11 **Interest Calculations** Suppose that you have a sum of money P in an interest-bearing account at a local bank (P stands for *present value*). If the bank pays you interest on the money at a rate of i percent per year and compounds the interest monthly, the amount of money that you will have in the bank after n months is given by the equation

$$F = P\left(1 + \frac{i}{1200}\right)^n \tag{8-5}$$

where F is the future value of the account and $\frac{i}{12}$ is the monthly percent-

age interest rate (the extra factor of 100 in the denominator converts the interest rate from percentages to fractional amounts). Write a MATLAB program that will read an initial amount of money P and an annual interest rate i and will calculate and write out a table showing the future value of the account every month for the next five years. The table should be written to an output file called `'interest'`. Be sure to properly label the columns of your table.

8.12 Write a program to read a set of integers from an input data file and locate the largest and smallest values within the data file. Print out the largest and smallest values, together with the lines on which they were found. Assume that you do not know the number of values in the file before the file is read.

8.13 Create a 400×400 element `double` array `x` and fill it with random data using function `rand`. Save this array to a MAT file `x1.dat` and then save it again to a second MAT file `x2.dat` using the `-compress` option. How do the sizes of the two files compare?

8.14 **Means** In Exercise 4.27, we wrote a MATLAB program that calculated the arithmetic mean (average), rms average, geometric mean, and harmonic mean for a set of numbers. Modify that program to read an arbitrary number of values from an input data file, and calculate the means of those numbers. To test the program, place the following values into an input data file and run the program on that file: 1.0, 2.0, 5.0, 4.0, 3.0, 2.1, 4.7, 3.0.

8.15 **Converting Radians to Degrees/Minutes/Seconds** Angles are often measured in degrees (°), minutes (′), and seconds (″), with 360 degrees in a circle, 60 minutes in a degree, and 60 seconds in a minute. Write a program that reads angles in radians from an input disk file and converts them into degrees, minutes, and seconds. Test your program by placing the following four angles expressed in radians into an input file and reading that file into the program: 0.0, 1.0, 3.141593, 6.0.

8.16 Create a data set in some other program on your computer (e.g., Microsoft Word, Microsoft Excel, a text editor). Copy the data set to the clipboard using the Windows or Unix copy function, and then use function `uiimport` to load the data set into MATLAB.

CHAPTER 9

Handle Graphics

Handle graphics is the name of a set of low-level graphics functions that control the characteristics of graphics objects generated by MATLAB. These functions are normally hidden inside M-files, but they are very important to the programmer, since they allow him or her to have fine control of the appearance of the plots and graphs that they generate. For example, it is possible to use handle graphics to turn on a grid on the *x*-axis only, or to choose a line color like orange, which is not supported by the standard `LineSpec` option of the `plot` command. In addition, handle graphics enable a programmer to created graphical user interfaces (GUIs) for programs, as is shown in Chapter 10.

This chapter introduces the structure of the MATLAB graphics system and explains how to control the properties of graphical objects to create a desired display.

9.1 The MATLAB Graphics System

The MATLAB graphics system is based on a hierarchical system of core **graphics objects**, each of which is known by a unique number called a **handle**. Each graphics object has special data known as **properties** associated with it, and modifying those properties will modify the behavior of the object. For example, a **`line`** is one type of graphics object. The properties associated with a line object include: *x*-data, *y*-data, color, line style, line width, and marker type, among others. Modifying any of these properties will change the way the line is displayed in a Figure Window.

Every component of a MATLAB graph is a graphical object. For example, each line, axis, and text string is a separate object with its own unique identifying number (handle) and characteristics. All graphical objects are arranged in a hierarchy with

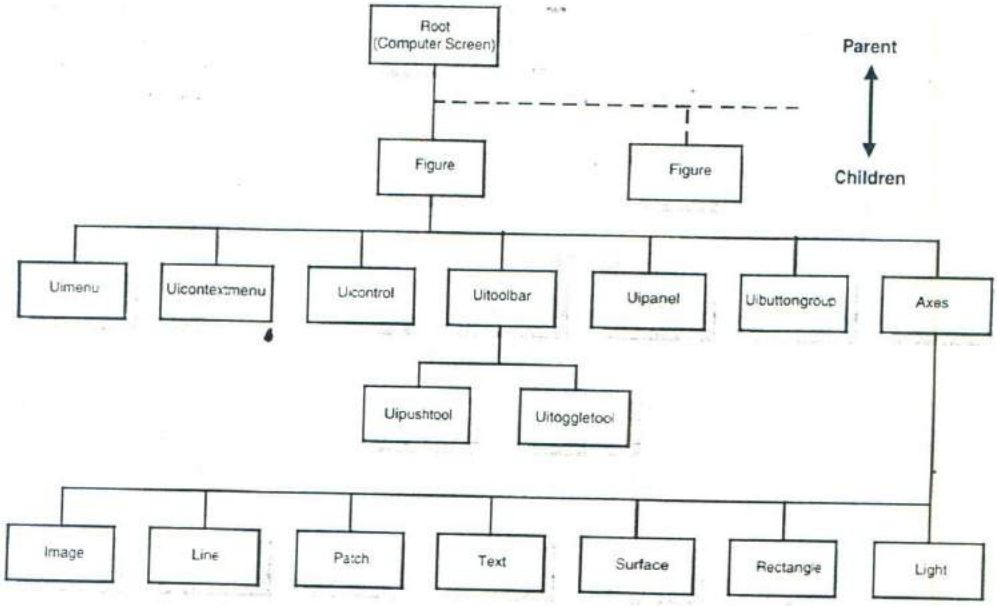


Figure 9.1 The hierarchy of handle graphics objects.

parent objects and **child objects**, as shown in Figure 9.1. When a child object is created, it inherits many of its properties from its parent.

The highest-level graphics object in MATLAB is the **`root`**, which can be thought of as the entire computer screen. The handle of the `root` object is always 0. It is created automatically when MATLAB starts up, and it is always present until the program is shut down. The properties associated with the root object are the defaults that apply to all MATLAB windows.

Under the root there can be one or more Figure Windows, or just **`figures`**. Each `figure` is a separate window on the computer screen that can display graphical data, and each figure has its own properties. The properties associated with a `figure` include color, color map, paper size, paper orientation, and pointer type, among others.

Each `figure` can contain seven types of objects: `uimenus`, `uicontextmenus`, `uicontrols`, `uitoolbars`, `uipanels`, `uibuttongroups`, and `axes`. `Uimenus`, `uicontextmenus`, `uicontrols`, `uitoolbars`, `uipanels`, and `uibuttongroups` are special graphics objects used to create graphical user interfaces—they are described in the following chapter. `Axes` are regions within a figure where data is actually plotted. There can be more than one set of axes in a single figure.

Each set of axes can contain as many `lines`, `text` strings, or `patches` as necessary to create the plot of interest.

9.2 Object Handles

Each graphics object has a unique name called a **handle**. The handle is a unique integer or real number that is used by MATLAB to identify the object. A handle is automatically returned by any function that creates a graphics object. For example, the function call

```
» hndl = figure;
```

creates a new figure and returns the handle of that figure in variable `hndl`. The handle of the root object is always 0, and the handle of each figure is normally a small positive integer, such as 1, 2, 3, . . . The handles of all other graphics objects are arbitrary floating-point numbers.

There are MATLAB functions available to get the handles of figures, axes, and other objects. For example, the function `gcf` returns the handle of the currently selected figure, `gca` returns the handle of the currently selected axes within the currently selected figure, and `gco` returns the handle of the currently selected object. These functions are discussed in more detail later in this chapter.

By convention, handles are usually stored in variables that begin with the letter h. This practice helps us to recognize handles in MATLAB programs.

9.3 Examining and Changing Object Properties

Object properties are special values associated with an object that control some aspect of how that object behaves. Each property has a **property name** and an associated value. The property names are strings that are typically displayed in mixed case with the first letter of each word capitalized, but MATLAB recognizes a property name regardless of the case in which it is written.

Changing Object Properties at Creation Time

When an object is created, all of its properties are automatically initialized to default values. These default values can be overridden at creation time by including `'PropertyName', value` pairs in the object creation function.[1]

[1] Examples of object creation functions include `figure`, which creates a new figure; `axes`, which creates a new set of axes within a figure; and `line`, which creates a line within a set of axes. High-level functions such as `plot` are also object-creation functions.

For example, we saw in Chapter 2 that the width of a line could be modified in the `plot` command as follows.

```
plot(x,y,'LineWidth',2);
```

This function overrides the default `LineWidth` property with the value 2 at the time the line object is created.

Changing Object Properties After Creation Time

The properties of any object can be examined at any time using the `get` function, and can be modified using the `set` function. The most common forms of the `get` function are

```
value = get(handle,'PropertyName');
value = get(handle);
```

where `value` is the value contained in the specified property of the object whose handle is supplied. If only the handle is included in the function call, then the function returns a structure array in which the field names are all of the properties of the object, and the field values are the property values.

The most common form of the `set` function is

```
set(handle,'PropertyName1',value1,...);
```

where there can be any number of `'PropertyName',value` pairs in a single function.

For example, suppose that we plotted the function $y(x) = x^2$ from 0 to 2 with the following statements:

```
x = 0:0.1:2;
y = x.^2;
hndl = plot(x,y);
```

The resulting plot is shown in Figure 9.2*a*. The handle of the plotted line is stored in `hndl`, and we can use it to examine or modify the properties of the line. The function `get(hndl)` will return all of the properties of this line in a structure, with each property name being an element of the structure.

```
» result = get(hndl)
result =
                Color: [0 0 1]
           EraseMode: 'normal'
           LineStyle: '-'
           LineWidth: 0.5000
              Marker: 'none'
          MarkerSize: 6
     MarkerEdgeColor: 'auto'
     MarkerFaceColor: 'none'
               XData: [1x21 double]
               YData: [1x21 double]
```

```
                ZData: [1x0 double]
        BeingDeleted: 'off'
        ButtonDownFcn: []
            Children: [0x1 double]
            Clipping: 'on'
            CreateFcn: []
            DeleteFcn: []
            BusyAction: 'queue'
        HandleVisibility: 'on'
            HitTest: 'on'
        Interruptible: 'on'
            Selected: 'off'
    SelectionHighlight: 'on'
                Tag: ''
                Type: 'line'
        UIContextMenu: []
            UserData: []
            Visible: 'on'
            Parent: 303.0004
        DisplayName: ''
        XDataMode: 'manual'
        XDataSource: ''
        YDataSource: ''
        ZDataSource: ''
```

Note that the current line width is 0.5 pixels and the current line style is a solid line. We can change the line width and the line style with the commands

```
» set(hndl,'LineWidth',4,'LineStyle','--')
```

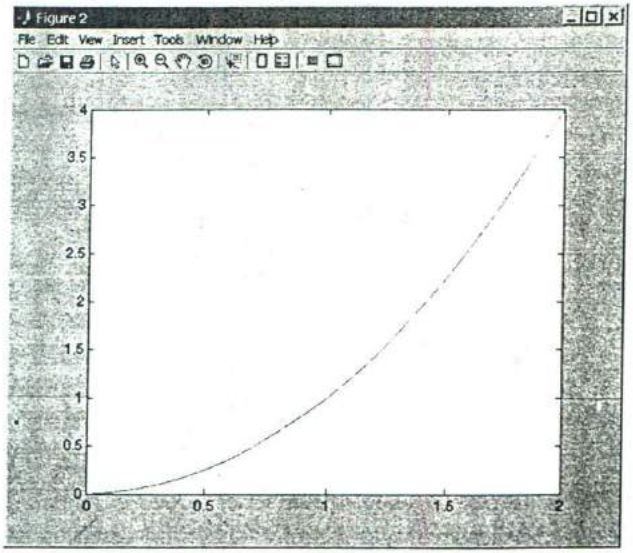
The plot after this command is issued is shown in Figure 9.2*b*.

The `get` and `set` functions are especially useful for programmers, because they can be directly inserted into MATLAB programs to modify a figure based on a user's input. As is shown in Chapter 10, these function are used extensively in GUI programming.

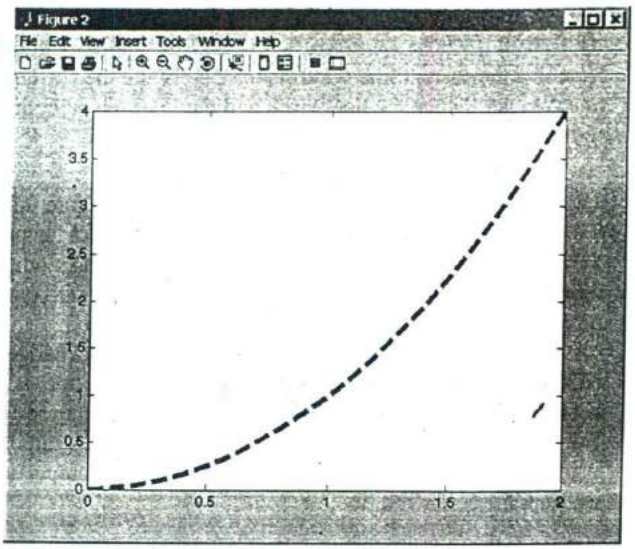
For the end user, however, it is often easier to change the properties of a MATLAB object interactively. The Property Editor is a GUI-based tool designed for this purpose. The Property Editor is started by first selecting the Edit Button ( ) on the figure toolbar and then clicking on the object that you want to modify with the mouse. Alternatively, the property editor can be started from the command line.

```
propedit(HandleList);
propedit;
```

For example, the following statements will create a plot containing the line $y = x^2$ over the range 0 to 2, and will open the Property Editor to allow the user to interactively change the properties of the line.



(a)



(b)

Figure 9.2 *(a)* Plot of the function $y = x^2$ using the default linewidth. *(b)* Plot of the function after modifying the `LineWidth` and `LineStyle` properties.

Color Figures These pages include color versions of selected figures from the text. These figures are included in color because the color illustrates some essential point about each figure that could not be displayed in greyscale.

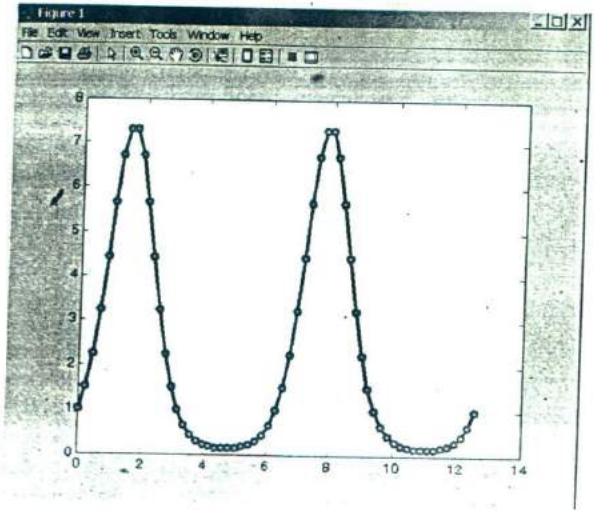


Figure 3.8 A plot illustrating the use of the `LineWidth` and `Marker` properties. (Page 124)

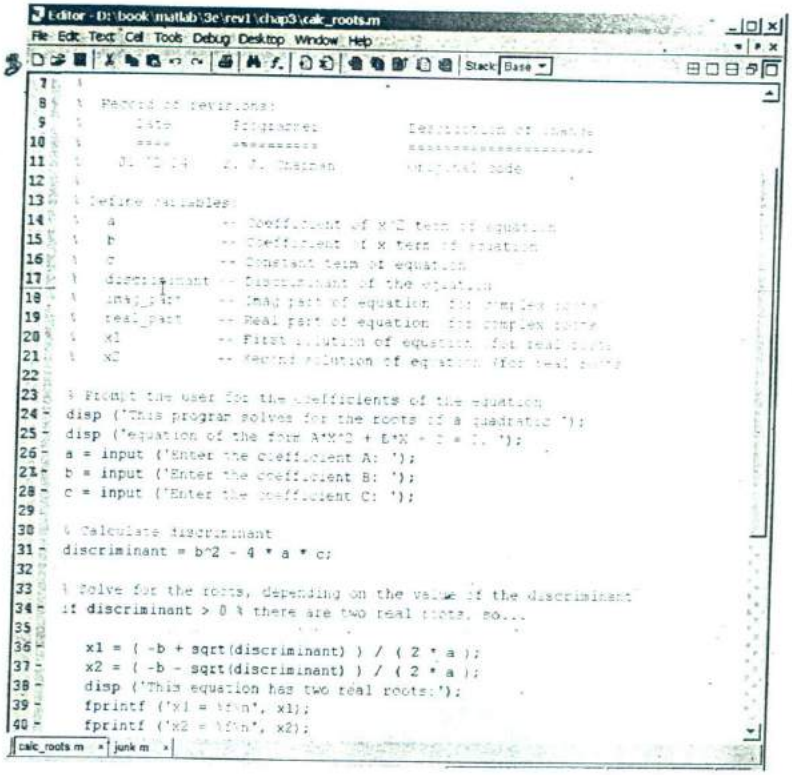
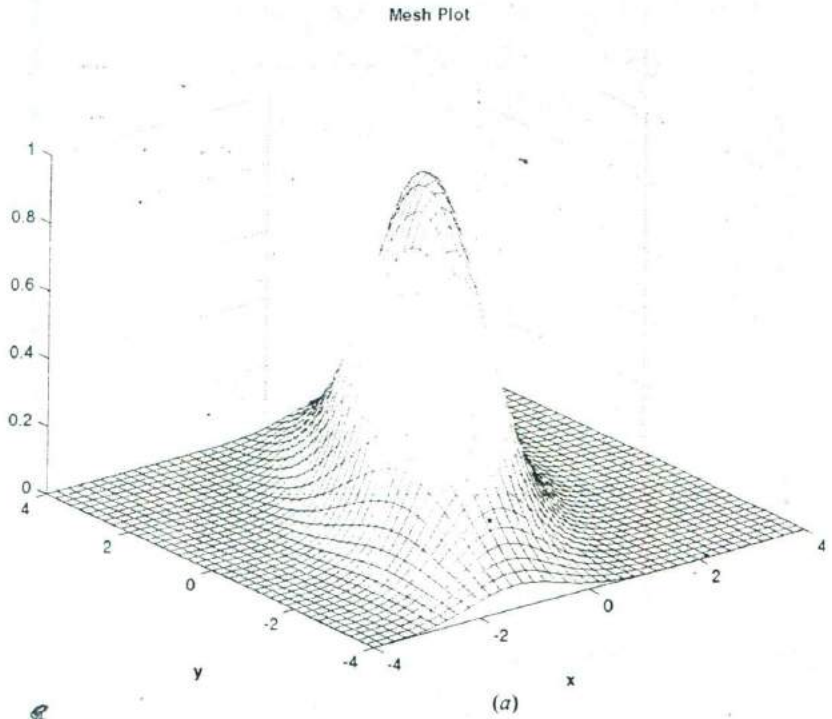
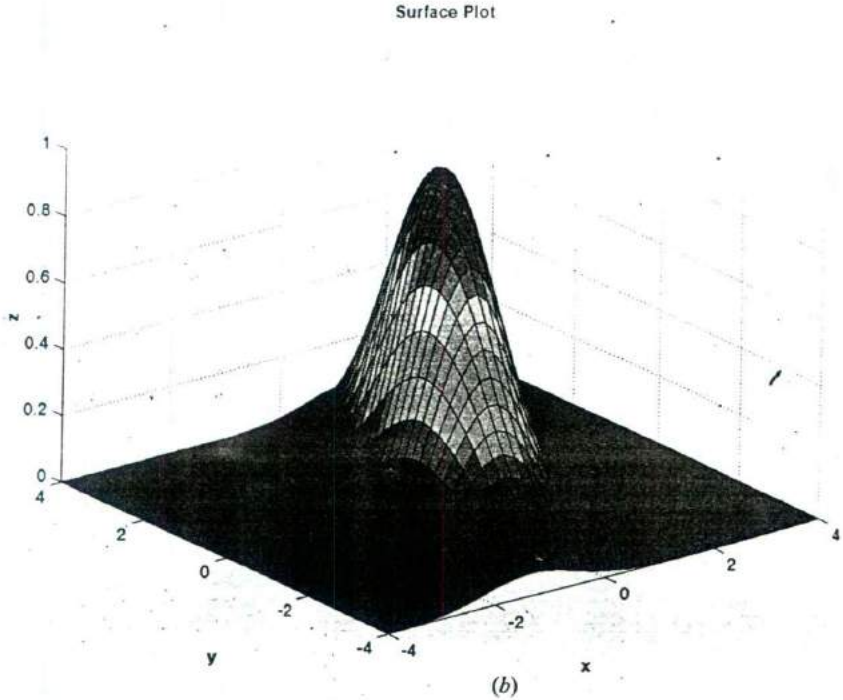


Figure 3.16 An Edit/Debug window with a MATLAB program loaded. (Page 136)

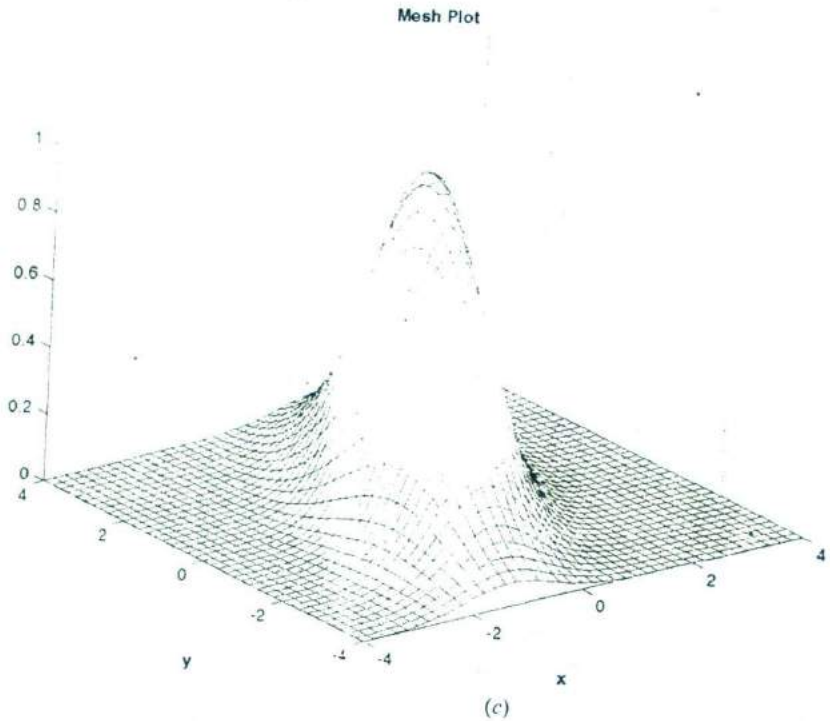


(*a*)

Figure 6.12 (*a*) A mesh plot of the function $z(x, y) = e^{-0.5[x^2+0.5(x-y)^2]}$. (*b*) A surface plot of the same function. (Pages 307–308)



(*b*)



(*c*)

Figure 6.12 (*continued*) (*c*) A contour plot of the same function.

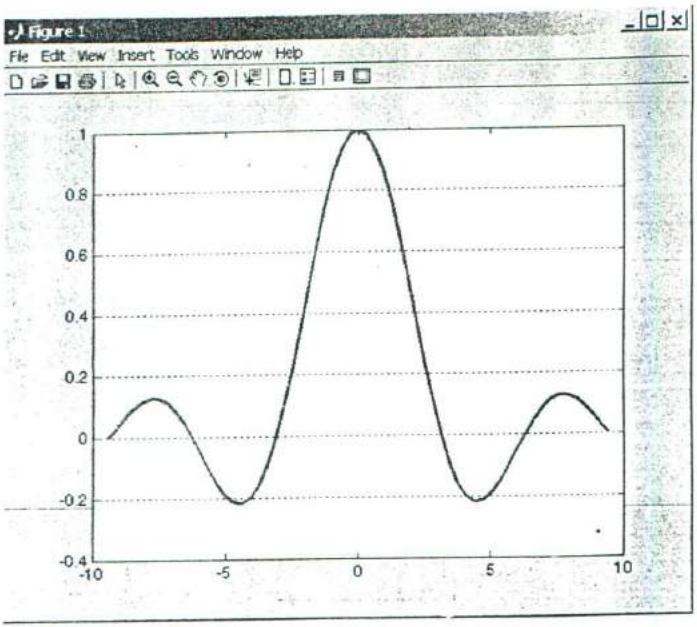


Figure 9.4 Plot of sinc x versus x. (Page 418)

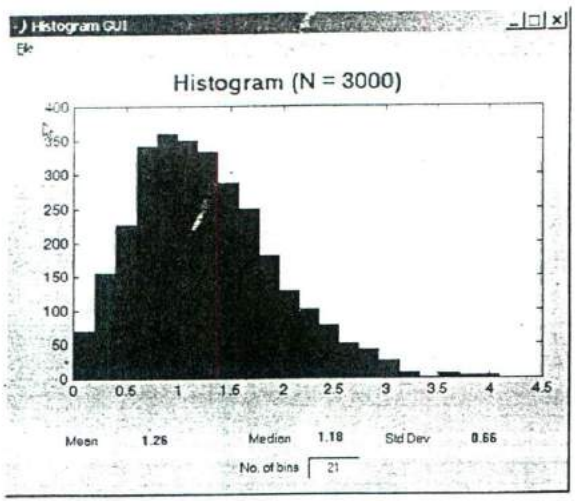


Figure 10.45 The GUI produced by program `HistGUI`. (Page 502)

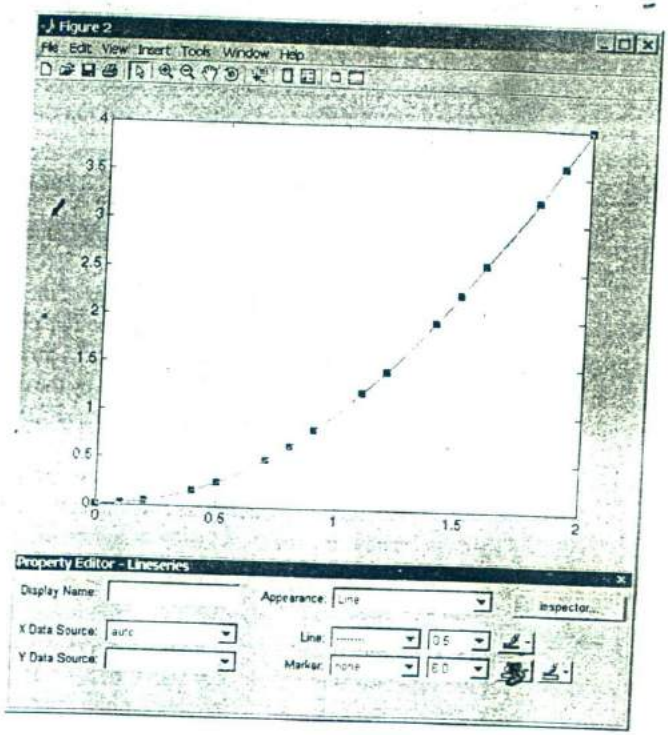


Figure 9.3 The Property Editor when editing a line object. Changes in style are immediately displayed on the figure as the object is edited.

```
figure(2);
x = 0:0.1:2;
y = x.^2;
hndl = plot(x,y);
propedit(hndl);
```

The Property Editor invoked by these statements is shown in Figure 9.3. The Property Editor contains a series of panes that vary depending on the type of object being modified.

Example 9.1—Using Low-Level Graphics Commands

The function sinc(x) is defined by the equation

$$\operatorname{sinc} x = \begin{cases} \dfrac{\sin x}{x} & x \neq 0 \\ 1 & x = 0 \end{cases} \tag{9-1}$$

Plot this function from $x = -3\pi$ to $x = 3\pi$. Use handle graphics functions to customize the plot as follows:

1. Make the figure background pink.
2. Use y-axis grid lines only (no x-axis grid marks).
3. Plot the function as a 2-point-wide solid orange line.

SOLUTION To create this graph, we need to plot the function sinc x from $x = -3\pi$ to $x = 3\pi$ using the `plot` function. The plot function will return a handle for the line that we can save and use later.

After plotting the line, we need to modify the color of the *figure* object, the grid status of the *axes* object, and the color and width of the *line* object. These modifications require us to have access to the handles of the `figure`, `axes`, and `line` objects. The handle of the `figure` object is returned by the `gcf` function, the handle of the `axes` object is returned by the `gca` function, and the handle of the `line` object is returned by the `plot` function that created it.

The low-level graphics properties that need to be modified can be found by referring to the on-line MATLAB Help Browser documentation, under the topic "Handle Graphics," They are the `'Color'` property of the current figure, the `'YGrid'` property of the current axes, and the `'LineWidth'` and `'Color'` properties of the line.

1. **State the problem.**
   Plot the function sinc x from $x = -3\pi$ to $x = 3\pi$ using a figure with a pink background, y-axis grid lines only, and a 2-point-wide solid orange line.

2. **Define the inputs and outputs.**
   There are no inputs to this program, and the only output is the specified figure.

3. **Describe the algorithm.**
   This program can be broken down into three major steps

   ```
   Calculate sinc(x)
   Plot sinc(x)
   Modify the required graphics object properties
   ```

   The first major step is to calculate sinc x from $x = -3\pi$ to $x = 3\pi$. This can be done with vectorized statements, but the vectorized statements will produce a `NaN` at $x = 0$, since the division of 0/0 is undefined. We must replace the `NaN` with a 1.0 before plotting the function. The detailed pseudocode for this step is:

   ```
   % Calculate sinc(x)
   x = -3*pi:pi/10:3*pi
   y = sin(x) ./ x

   % Find the zero value and fix it up. The zero is
   % located in the middle of the x array.
   ```

```
index = fix(length(y)/2) + 1
y(index) = 1
```

Next, we must plot the function, saving the handle of the resulting line for further modifications. The detailed pseudocode for this step is:

```
hndl = plot(x,y);
```

Now we must use handle graphics commands to modify the figure background, *y*-axis grid, and line width and color. Remember that the figure handle can be recovered with the function `gcf`, and the axis handle can be recovered with the function `gca`. The color pink can be created with the RGB vector `[1 0.8 0.8]`, and the color orange can be created with the RGB vector `[1 0.5 0]`. The detailed pseudocode for this step is:

```
set(gcf,'Color',[1 0.8 0.8])
set(gca,'YGrid','on')
set(hndl,'Color',[1 0.5 0],'LineWidth',2)
```

4. **Turn the algorithm into MATLAB statements.**
The final MATLAB program is shown below.

```
% Script file: plotsinc.m
%
% Purpose:
%   This program illustrates the use of handle graphics
%   commands by creating a plot of sinc(x) from -3*pi to
%   3*pi, and modifying the characteristics of the figure,
%   axes, and line using the "set" function.
%
% Record of revisions:
%     Date          Programmer        Description of change
%     ====          ==========        =====================
%   01/24/04     S. J. Chapman      Original code
%
% Define variables:
%   hndl          -- Handle of line
%   x             -- Independent variable
%   y             -- sinc(x)

% Calculate sinc(x)
x = -3*pi:pi/10:3*pi;
y = sin(x) ./ x;

% Find the zero value and fix it up. The zero is
% located in the middle of the x array.
index = fix(length(y)/2) + 1;
y(index) = 1;
```

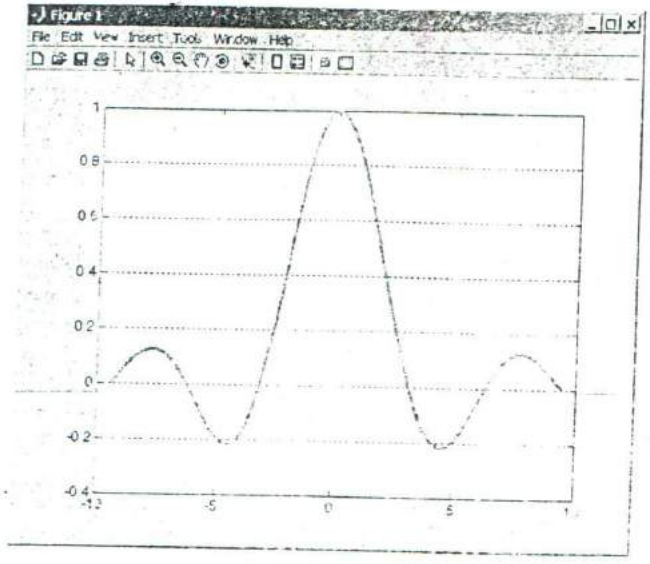


Figure 9.4 Plot of sine x versus x. A color version of this plot appears in the color insert.

```
% Plot the function.
hndl = plot(x,y);

% Now modify the figure to create a pink background,
% modify the axis to turn on y-axis grid lines, and
% modify the line to be a 2-point wide orange line.
set(gcf,'Color',[1 0.8 0.8]);
set(gca,'YGrid','on');
set(hndl,'Color',[1 0.5 0],'LineWidth',2);
```

5. **Test the program.**
   Testing this program is very simple—we just execute it and examine the resulting plot. The plot created is shown in Figure 9.4, and it does have the characteristics that we wanted.

9.4 Using set to List Possible Property Values

The `set` function can be used to provide lists of possible property values. If a `set` function call contains a property name but not a corresponding value, `set` returns a list of all of the legal choices for that property. For example, the command `set(hndl,'LineStyle')` will return a list of all legal line styles with the default choice in brackets:

```
» set(hndl,'LineStyle')
ans =
    '-'
    '--'
    ':'
    '-.'
    'none'
```

This function shows that the legal line styles are '-', '--', ':', '-.', and 'none', with the first choice as the default.

If the property does not have a fixed set of values, MATLAB returns an empty cell array:

```
» set(hndl,'LineWidth')
ans =
    {}
```

The function set(hndl) will return all of the possible choices for all of the properties of an object.

```
» xxx = set(hndl)
xxx =
                Color: {}
          EraseMode: {4x1 cell}
          LineStyle: {5x1 cell}
          LineWidth: {}
             Marker: {14x1 cell}
         MarkerSize: {}
    MarkerEdgeColor: {2x1 cell}
    MarkerFaceColor: {2x1 cell}
              XData: {}
              YData: {}
              ZData: {}
        ButtonDownFcn: {}
             Children: {}
             Clipping: {2x1 cell}
           CreateFcn: {}
           DeleteFcn: {}
         BusyAction: {2x1 cell}
      HandleVisibility: {3x1 cell}
              HitTest: {2x1 cell}
        Interruptible: {2x1 cell}
             Selected: {2x1 cell}
  SelectionHighlight: {2x1 cell}
                 Tag: {}
       UIContextMenu: {}
             UserData: {}
```

```
           Visible: {2x1 cell}
            Parent: {}
       DisplayName: {}
        XDataMode: {2x1 cell}
       XDataSource: {}
       YDataSource: {}
       ZDataSource: {}
```

Any of the items in this list can be expanded to see the available list of options.

```
» xxx.EraseMode
ans =
    'normal'
    'background'
    'xor'
    'none'
```

9.5 User-Defined Data

In addition to the standard properties defined for a GUI object, a programmer can define special properties to hold program-specific data. These extra properties are a convenient way to store any kind of data that the programmer might wish to associate with the GUI object. Any amount of any type of data can be stored and used for any purpose.

User-defined data is stored in a manner similar to standard properties. Each data item has a name and a value. Data values are stored in an object with the `setappdata` function, and retrieved from the object using the `getappdata` function.

The general form of `setappdata` is

```
setappdata(hndl,'DataName',DataValue);
```

where `hndl` is the handle of the object to store the data into, `'DataName'` is the name given to the data, and `DataValue` is the value assigned to that name. Note that the data value can be either numeric or a character string.

For example, suppose that we wanted to define two special data values: one containing the number of errors that have occurred on a particular figure, and the other containing a string describing the last error detected. Such data values could be given the names `'ErrorCount'` and `'LastError'`. If we assume that `h1` is the handle of the figure, then command to create these data items and initialize them would be:

```
setappdata(h1,'ErrorCount',0);
setappdata(h1,'LastError','No error');
```

Application data can be retrieved at any time using the function `getappdata`. The two forms of `getappdata` are

```
value = getappdata(hndl,'DataName');
struct = getappdata(hndl);
```

where `hndl` is the handle of the object containing the data, and `'DataName'` is the name of the data to be retrieved. If a `'DataName'` is specified, then the value associated with that data name will be returned. If it is not specified, then *all* user-defined data associated with that object will be returned in a structure. The names of the data items will be structure element names in the returned structure.

For the example given above, `getappdata` will produce the following results:

```
» value = getappdata(h1,'ErrorCount')
value =
     0
» struct = getappdata(h1)
struct =
     ErrorCount: 0
     LastError: 'No error'
```

The functions associated with user-defined data are summarized in Table 9.1.

Table 9.1 Functions for Manipulating User-Defined Data

Function	Description
`setappdata(hndl,'DataName',DataValue)`	Stores `DataValue` in an item named `'DataName'` within the object specified by the handle `hndl`.
`value = getappdata(hndl,'DataName')` `struct = getappdata(hndl)`	Retrieves user-defined data from the object specified by the handle `hndl`. The first form retrieves the value associated with `'DataName'` only, and the second form retrieves all user-defined data.
`isappdata(hndl,'DataName')`	A logical function that returns a 1 if `'DataName'` is defined within the object specified by the handle `hndl`, and 0 otherwise.
`rmappdata(hndl,'DataName')`	Removes the user-defined data item named `'DataName'` from the object specified by the handle `hndl`.

9.6 Finding Objects

Each new graphics object that is created has its own handle, and that handle is returned by the creating function. If you intend to modify the properties of an object that you create, then it is a good idea to save the handle for later use with `get` and `set`.

Good Programming Practice

If you intend to modify the properties of an object that you create, save the handle of that object for later use with `get` and `set`.

However, sometimes we might not have access to the handle. Suppose that we lost a handle for some reason. How can we examine and modify the graphics objects?

MATLAB provides four special functions to help find the handles of objects.

- `gcf`—Returns the handle of the current *figure*.
- `gca`—Returns the handle of the current *axes* in the current *figure*.
- `gco`—Returns the handle of the current *object*.
- `findobj`—Finds a graphics object with a specified property value.

The function `gcf` returns the handle of the current figure. If no figure exists, `gcf` *will create one* and return its handle. The function `gca` returns the handle of the current axes within the current figure. If no figure exists or if the current figure exists but contains no axes, `gca` *will create a set of axes* and return its handle. The function `gco` has the form

```
h_obj = gco;
h_obj = gco(h_fig);
```

where `h_obj` is the handle of the object and `h_fig` is the handle of a figure. The first form of this function returns the handle of the *current object in the current figure*, while the second form of the function returns the handle of the *current object in a specified figure*.

The current object is defined as the last object clicked on with the mouse. This object can be any graphics object except the root. There will not be a current object in a figure until a mouse click has occurred within that figure. Before the first mouse click, function `gco` will return an empty array `[]`. Unlike `gcf` and `gca`, `gco` does not create an object if it does not exist.

Once the handle of an object is known, we can determine the type of the object by examining its `'Type'` property. The `'Type'` property will be a character string, such as `'figure'`, `'line'`, `'text'`, and so forth.

```
h_obj = gco;
type = get(h_obj,'Type')
```

The easiest way to find an arbitrary MATLAB object is with the `findobj` function. The basic form of this function is

```
hndls = findobj('PropertyName1',value1,...)
```

This command starts at the root object, and searches the entire tree for all objects that have the specified values for the specified properties. Note that multiple property/value pairs may be specified, and `findobj` returns only the handles of objects that match *all* of them.

For example, suppose that we have created Figures 1 and 3. Then the function `findobj('Type','figure')` will return the results:

```
» h_fig = findobj('Type','figure')
h_fig =
      3
      1
```

This form of the `findobj` function is very useful, but it can be slow, since it must search through the entire object tree to locate any matches. If you must use an object multiple times, only make one call to `findobj` and save the handle for reuse.

Restricting the number of objects that must be searched can increase the execution speed of this function. This can be done with the following form of the function:

```
hndls = findobj(Srchhndls,'PropertyName1',value1,...)
```

Here, only the handles listed in array `Srchhndls` and their children will be searched to find the object. For example, suppose that you wanted to find all of the dashed lines in Figure 1. The command to do this would be:

```
hndls = findobj(1,'Type','line','LineStyle','--');
```

Good Programming Practice

If possible, restrict the scope of your searches with `findobj` to make them faster.

9.7 Selecting Objects with the Mouse

Function `gco` returns the handle of the current object, which is the last object clicked on by the mouse. Each object has a **selection region** associated with it, and any mouse click within that selection region is assumed to be a click on that object. This is very important for thin objects like lines or points—the selection region allows the user to be slightly sloppy in mouse position and still select the line. The width and shape of the selection region varies for different types of objects. For instance, the selection region for a line is five pixels on either side of the line, while the selection region for a surface, patch, or text object is the smallest rectangle that can contain the object.

The selection region for an axes object is the area of the axes plus the area of the titles and labels. However, lines or other objects inside the axes have a higher priority, so to select the axes you must click on a point within the axes that is not near lines or text. Clicking on a figure outside of the axes region will select the figure itself.

What happens if a user clicks on a point that has two or more objects, such as the intersection of two lines? The answer depends on the **stacking order** of the objects. The stacking order is the order in which MATLAB selects objects. This order is specified by the order of the handles listed in the `'Children'` property of a figure. If a click is in the selection region of two or more objects, the one with the highest position in the `'Children'` list will be selected.

MATLAB includes a function called `waitforbuttonpress` that is sometimes used when selecting graphics objects. The form of this function is:

```
k = waitforbuttonpress
```

When this function is executed, it halts the program until either a key is pressed or a mouse button is clicked. The function returns 0 if it detects a mouse button click or 1 if it detects a key press.

The function can be used to pause a program until a mouse click occurs. After the mouse click occurs, the program can recover the handle of the selected object using the `gco` function.

Example 9.2—Selecting Graphics Objects

The program shown below explores the properties of graphics objects and incidentally shows how to select objects using `waitforbuttonpress` and `gco`. The program allows objects to be selected repeatedly until a key press occurs.

```
% Script file: select_object.m
%
% Purpose:
%   This program illustrates the use of waitforbuttonpress
%   and gco to select graphics objects. It creates a plot
%   of sin(x) and cos(x), and then allows a user to select
%   any object and examine its properties. The program
%   terminates when a key press occurs.
%
% Record of revisions:
%      Date          Programmer        Description of change
%      ====          ==========        =====================
%   01/25/04     S. J. Chapman      Original code
%
% Define variables:
%   details        -- Object details
%   h1             -- handle of sine line
%   h2             -- handle of cosine line
%   handle         -- handle of current object
```

```
%   k           -- Result of waitforbuttonpress
%   type        -- Object type
%   x           -- Independent variable
%   y1          -- sin(x)
%   y2          -- cos(x)
%   yn          -- Yes/No

% Calculate sin(x) and cos(x)
x = -3*pi:pi/10:3*pi;
y1 = sin(x);
y2 = cos(x);

% Plot the functions.
h1 = plot(x,y1);
set(h1,'LineWidth',2);
hold on;
h2 = plot(x,y2);
set(h2,'LineWidth',2,'LineStyle',':','Color','r');
title('\bfPlot of sin \itx \rm\bf and cos \itx');
xlabel('\bf\itx');
ylabel('\bfsin \itx \rm\bf and cos \itx');
legend('sine','cosine');
hold off;

% Now set up a loop and wait for a mouse click.
k = waitforbuttonpress;

while k == 0

   % Get the handle of the object
   handle = gco;

   % Get the type of this object.
   type = get(handle,'Type');

   % Display object type
   disp (['Object type = ' type '.']);

   % Do we display the details?
   yn = input('Do you want to display details? (y/n) ','s');

   if yn == 'y'
      details = get(handle);
      disp(details);
   end

   % Check for another mouse click
   k = waitforbuttonpress;
end
```

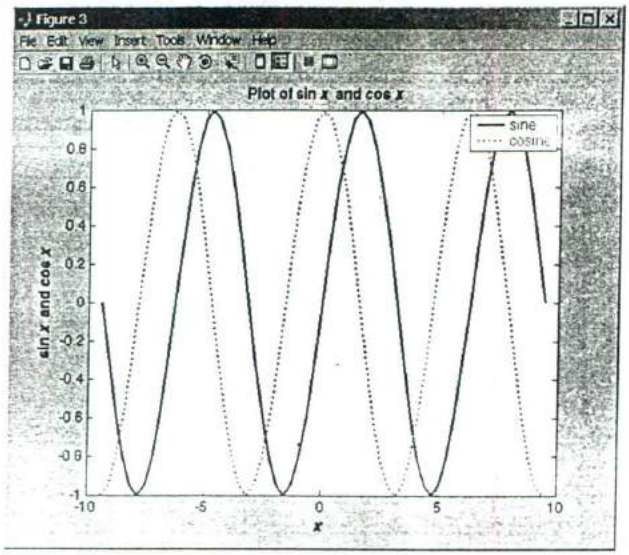


Figure 9.5 Plot of sin x and cos x.

When this program is executed, it produces the plot shown in Figure 9.5. Experiment by clicking on various objects on the plot, and seeing the resulting characteristics.

◀

9.8 Position and Units

Many MATLAB objects have a `'position'` property, which specifies the size and position of the object on the computer screen. This property differs slightly for different kinds of objects, as described below.

Positions of figure Objects

The `'position'` property for a figure specifies the location of that figure on the computer screen using a 4-element row vector. The values in this vector are `[left bottom width height]`, where `left` is the leftmost edge of the figure, `bottom` is the bottom edge of the figure, `width` is the width of the figure, and `height` is the height of the figure. These position values are in the units specified in the `'Units'` property for the object. For example, the position and units associated with a the current figure can be found as follows:

```
» get(gcf,'Position')
ans =
   176   204   672   504
» get(gcf,'Units')
ans =
pixels
```

This information specifies that the lower left corner of the current figure window is 176 pixels to the right and 204 pixels above the lower left corner of the screen, and the figure is 672 pixels wide by 504 pixels high. This is the drawable region of the figure, excluding borders, scrollbars, menus, and the figure title area.

The `'units'` property of a figure defaults to pixels, but it can be inches, centimeters, points, characters, or normalized coordinates. Pixels are screen pixels, which are the smallest rectangular shape that can be drawn on a computer screen. Typical computer screens will be at least 640 pixels wide by 480 pixels high, and screens can have more than 1000 pixels in each direction. Since the number of pixels varies from computer screen to computer screen, the size of an object specified in pixels will also vary.

Normalized coordinates are coordinates in the range 0 to 1, where the lower left corner of the screen is at (0, 0) and the upper right corner of the screen is at (1, 1). If an object position is specified in normalized coordinates, it will appear in the same relative position on the screen regardless of screen resolution. For example, the following statements create a figure and place it into the upper left quadrant of the screen on any computer, regardless of screen size.[2]

```
h1 = figure(1)
set(h1,'units','normalized','position',[0 .5 .5 .45])
```

Good Programming Practice

If you would like to place a window in a specific location, it is easier to place the window at the desired location using normalized coordinates; the results will be the same regardless of the computer's screen resolution.

Positions of `axes` and `uicontrol` Objects

The position of `axes` and `uicontrol` objects is also specified by a 4-element vector, but the object position is specified relative to the lower left-hand corner of the *figure* instead of the position of the screen. In general, the `'Position'` property of a child object is relative to the position of its parent.

[2]The normalized height of this figure is reduced to 0.45 to allow room for the Figure title and menu bar, both of which are above the drawing area.

By default, the positions of axes objects are specified in *normalized* units within a figure, with (0, 0) representing the lower left-hand corner of the figure, and (1, 1) representing the upper right-hand corner of the figure.

Positions of text Objects

Unlike other objects, `text` objects have a position property containing only two or three elements. These elements correspond to the x, y, and z values of the text object *within* an `axes` object. Note that these values are in the units being displayed on the axes themselves.

The position of the text object with respect to the specified point is controlled by the object's `HorizontalAlignment` and `VerticalAlignment` properties. The `HorizontalAlignment` can be `{Left}`, `Center`, or `Right`, and the `VerticalAlignment` can be `Top`, `Cap`, `{Middle}`, `Baseline`, or `Bottom`.

The size of `text` objects is determined by the font size and the number of characters being displayed, so there are no height and width values associated with them.

Example 9.3—Positioning Objects within a Figure

As we mentioned earlier, axis positions are defined relative to the lower left-hand corner of the frame that they are contained in, while text object positions are defined within axes in the data units being displayed on the axes.

To illustrate the positioning of graphics objects within a figure, we will write a program that creates two overlapping sets of axes within a single figure. The first set of axes will display $\sin x$ versus x, and will have a text comment attached to the display line. The second set of axes will display $\cos x$ versus x, and will have a text comment in the lower left hand corner.

A program to create the figure is shown below. Note that we are using the `figure` function to create an empty figure, and then two `axes` functions to create the two sets of axes within the figure. The position of the `axes` functions is specified in normalized units within the figure, so the first set of axes, which starts at (0.05, 0.05), is in the lower left-hand corner of the figure, and the second set of axes, which starts at (0.45, 0.45), is in the upper right-hand corner of the figure. Each set of axes has the appropriate function plotted on it.

The first `text` object is attached to the first set of axes at position $(-\pi, 0)$, which is a point on the curve. The `'HorizontalAlignment','right'` property is selected, so the *attachment point* $(-\pi, 0)$ is on the *right hand side* of the text string. As a result, the text appears to the *left* of the of the attachment point in the final figure. (This can be confusing for new programmers!)

The second `text` object is attached to the second set of axes at position $(-7.5, -0.9)$, which is near the lower left-hand corner of the axes. This string uses the default horizontal alignment, which is `'left'`, so the attachment point $(-7.5, -0.9)$ is on the *left hand side* of the text string. As a result, the text appears to the right of the of the attachment point in the final figure.

```
% Script file: position_object.m
%
% Purpose:
%   This program illustrates the positioning of graphics
%   graphics objects. It creates a figure, and then places
%   two overlapping sets of axes on the figure. The first
%   set of axes is placed in the lower left hand corner of
%   the figure, and contains a plot of sin(x). The second
%   set of axes is placed in the upper right hand corner of
%   the figure, and contains a plot of .cos(x). Then two
%   text strings are added to the axes, illustrating the
%   positioning of text within axes.
%
% Record of revisions:
%      Date          Programmer          Description of change
%      ====          ==========          =====================
%    01/25/04     S. J. Chapman        Original code
%
% Define variables:
%   h1              -- Handle of sine line
%   h2              -- Handle of cosine line
%   ha1             -- Handle of first axes
%   ha2             -- Handle of second axes
%   x               -- Independent variable
%   y1              -- sin(x)
%   y2              -- cos(x)

% Calculate sin(x) and cos(x)
x = -2*pi:pi/10:2*pi;
y1 = sin(x);
y2 = cos(x);

% Create a new figure
figure;

% Create the first set of axes and plot sin(x).
% Note that the position of the axes is expressed
% in normalized units.
ha1 = axes('Position',[.05 .05 .5 .5]);
h1 = plot(x,y1);
set(h1,'LineWidth',2);
title('\bfPlot of sin \itx');
xlabel('\bf\itx');
ylabel('\bfsin \itx');
axis([-8 8 -1 1]);
```

```
% Create the second set of axes and plot cos(x).
% Note that the position of the axes is expressed
% in normalized units.
ha2 = axes('Position',[.45 .45 .5 .5]);
h2 = plot(x,y1);
set(h2,'LineWidth',2,'Color','r','LineStyle','--');
title('\bfPlot of cos \itx');
xlabel('\bf\itx');
ylabel('\bfsin \itx');
axis([-8 8 -1 1]);

% Create a text string attached to the line on the first
% set of axes.
axes(ha1);
text(-pi,0.0,'sin(x)\rightarrow','HorizontalAlignment','right');

% Create a text string in the lower left hand corner
% of the second set of axes.
axes(ha2);
text(-7.5,-0.9,'Test string 2');
```

When this program is executed, it produces the plot shown in Figure 9.6. You should execute this program again on your computer, changing the size and/or location of the objects being plotted, and observing the results.

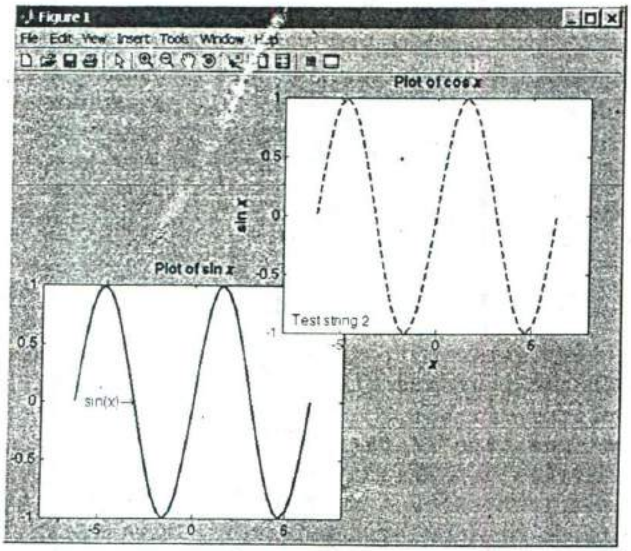


Figure 9.6 The output of program `position_object`.

9.9 Printer Positions

The 'Position' and 'Units' properties specify the location of a figure on the *computer screen*. There are also five other properties that specify the location of a figure on a sheet of paper *when it is printed*. These properties are summarized in Table 9.2.

For example, to set a plot to print out in landscape mode, on A4 paper, in normalized units, we could set the following properties:

```
set(hndl,'PaperType','A4')
set(hndl,'PaperOrientation','landscape')
set(hndl,'PaperUnits','normalized');
```

9.10 Default and Factory Properties

MATLAB assigns default properties to each object when it is created. If those properties are not what you want, then you must use set to select the desired values. If you wanted to change a property in every object that you create, this process could become very tedious. For those cases, MATLAB allows you to modify the default property itself, so that all objects will inherit the correct value of the property when they are created.

When a graphics object is created, MATLAB looks for a default value for each property by examining the object's parent. If the parent sets a default value, that value

Table 9.2 Printing-Related Figure Properties

Option	Description
PaperUnits	Units for paper measurements: [{inches} | centimeters | normalized | points]
PaperOrientation	[{portrait} | landscape]
PaperPosition	A position vector of the form [left, bottom, width, height], where all units are as specified in PaperUnits
PaperSize	A two-element vector containing the power size, for example [8.5 11]
PaperType	Sets paper type. Note that setting this property automatically updates the PaperSize property. [{usletter} | uslegal | A0 | A1 | A2 | A3 | A4 | A5 | B0 | B1 | B2 | B3 | B4 | B5 | arch-A | arch-B | arch-C | arch-D | arch-E | A | B | C | D | E | tabloid | <custom>]

is used. If not, MATLAB examines the parent's parent to see if that object sets a default value, and so on back to the root object. MATLAB uses the *first* default value that it encounters when working back up the tree.

Default properties may be set at any point in the graphics object hierarchy that is *higher* than the level at which the object is created. For example, a default `figure` color would be set in the `root` object, and then all figures created after that time would have the new default color. On the other hand, a default `axes` color could be set in either the `root` object or the `figure` object. If the default `axes` color is set in the `root` object, it will apply to all new axes in all figures. If the default `axes` color is set in the `figure` object, it will apply to all new axes in the current figure only.

`set(0,'DefaultFigureColor','y')`	Yellow figure background—all new figures.
`set(0,'DefaultAxesColor','r')`	Red axes background—all new axes in all figures.
`set(gcf,'DefaultAxesColor','r')`	Red axes background—all new axes in current figure only.
`set(gca,'DefaultLineLineStyle','--')`	Set default line style to dashed, in current axes only.

Default values are set using a string consisting of `'Default'` followed by the object type and the property name. Thus the default figure color would be set with the property `'DefaultFigureColor'` and the default axes color would be set with the property `'DefaultAxesColor'`. Some examples of setting default values are shown below:

If you are working with existing objects, it is always a good idea to restore them to their existing condition after they are used. *If you change the default properties of an object in a function, save the original values and restore them before exiting the function.* For example, suppose that we wish to create a series of figures in normalized units. We could save and restore the original units as follows:

```
saveunits = get(0,'DefaultFigureUnits');
set(0,'DefaultFigureUnits','normalized');
...
<MATLAB statements>
...
set(0,'DefaultFigureUnits',saveunits);
```

If you want to customize MATLAB to use different default values at all times, then you should set the defaults in the `root` object every time MATLAB starts up. The easiest way to do this is to place the default values into the `startup.m` file, which is automatically executed every time MATLAB starts. For exam-

ple, suppose you always use A4 paper and you always want a grid displayed on your plots. Then you could set the following lines into `startup.m`:

```
set(0,'DefaultFigurePaperType','A4');
set(0,'DefaultFigurePaperUnits','centimeters');
set(0,'DefaultAxesXGrid','on');
set(0,'DefaultAxesYGrid','on');
set(0,'DefaultAxesZGrid','on');
```

There are three special value strings that are used with handle graphics: `'remove'`, `'factory'`, and `'default'`. If you have set a default value for a property, the `'remove'` value will remove the default that you set. For example, suppose that you set the default figure color to yellow:

```
set(0,'DefaultFigureColor','y');
```

The following function call will cancel this default setting and restore the previous default setting.

```
set(0,'DefaultFigureColor','remove');
```

The string `'factory'` allows a user to temporarily override a default value and use the original MATLAB default value instead. For example, the following figure is created with the factory default color despite a default color of yellow being previously defined.

```
set(0,'DefaultFigureColor','y');
figure('Color','factory')
```

The string `'default'` forces MATLAB to search up the object hierarchy until it finds a default value for the desired property. It uses the first default value that it finds. If it fails to find a default value, it uses the factory default value for that property. This use is illustrated below:

```
% Set default values
set(0,'DefaultLineColor','k');   % root default = black
set(gcf,'DefaultLineColor','g'); % figure default = green

% Create a line on the current axes. This line is green.
hndl = plot(randn(1,10));
set(hndl,'Color','default');
pause(2);

% Now clear the figure's default and set the line color to the new
% default. The line is now black.
set(gcf,'DefaultLineColor','remove');
set(hndl,'Color','default');
```

9.11 Graphics Object Properties

There are hundreds of different graphic object properties, far too many to discuss in detail here. The best place to find a complete list of graphics object properties is in the Help Browser distributed with MATLAB.

We have mentioned a few of the most important properties for each type of graphic object as we have needed them (`'LineStyle,' 'Color'`, and so forth). A complete set of properties is given in the MATLAB Help Browser documentation under the descriptions of each type of object.

9.12 Summary

Every element of a MATLAB plot is a graphics object. Each object is identified by a unique handle, and each object has many properties associated with it which affect the way the object is displayed.

MATLAB objects are arranged in a hierarchy with **parent objects** and **child objects**. When a child object is created, it inherits many of its properties from its parent.

The highest-level graphics object in MATLAB is the `root`, which can be thought of as the entire computer screen. Under the root there can be one or more Figure Windows. Each `figure` is a separate window on the computer screen that can display graphical data, and each figure has its own properties.

Each `figure` can contain seven types of objects: `uimenus`, `uicontextmenus`, `uicontrols`, `uitoolbars`, `uipanels`, `uibuttongroups`, and `axes`. `Uimenus`, `uicontextmenus`, `uicontrols`, `uitoolbars`, `uipanels`, and `uibuttongroups` are special graphics objects used to create graphical user interfaces—they are described in the following chapter. `Axes` are regions within a figure where data is actually plotted. There can be more than one set of axes in a single figure.

Each set of axes can contain as many `lines`, `text strings`, or `patches` as necessary to create the plot of interest.

The handles of the current figure, current axes, and current object may be recovered with the `gcf`, `gca`, and `gco` functions, respectively. The properties of any object may be examined and modified using the `get` and `set` functions.

There are literally hundreds of properties associated with MATLAB graphics functions, and the best place to find the details of these of these functions is the MATLAB on-line documentation.

Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB handle graphics.

1. If you intend to modify the properties of an object that you create, save the handle of that object for later use with `get` and `set`.

2. If possible, restrict the scope of your searches with `findobj` to make them faster.
3. If you would like to place a window in a specific location, it is easier to place the window at the desired location using normalized coordinates, and the results will be the same regardless of the computer's screen resolution.

MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

`axes`	Creates a new axes/makes axes current.
`figure`	Creates a new figure makes figure current.
`findobj`	Finds an object based on one or more property values.
`gca`	Get handle of current axes.
`gcf`	Get handle of current figure.
`gco`	Get handle of current object.
`get`	Gets object properties.
`getappdata`	Gets user-defined data in an object.
`isappdata`	Tests to determine whether an object contains user-defined data with the specified name.
`rmappdata`	Removes user-defined data from an object.
`set`	Sets object properties.
`setappdata`	Stores user-defined data in an object.
`waitforbuttonpress`	Pauses program, waiting for a mouse click or keyboard input.

9.13 Exercises

9.1 What is meant by the term "handle graphics?" Sketch the hierarchy of MATLAB graphics objects.

9.2 Use the MATLAB Help Browser to learn about the `Name` and `NumberTitle` properties of a `figure` object. Create a figure containing a plot of the function $y(x) = e^x$ for $-2 \leq x \leq 2$. Change the properties mentioned above to suppress the figure number and to add the title "Plot Window" to the figure.

9.3 Write a program that modifies the default figure color to orange and the default line width to 3.0 points. Then create a figure plotting the ellipse defined by the equations

$$\begin{aligned} x(t) &= 10\cos t \\ y(t) &= 6\sin t \end{aligned} \tag{9-1}$$

from $t = 0$ to $t = 2\pi$. What color and width is the resulting line?

9.4 Use the MATLAB Help Browser to learn about the `CurrentPoint` property of an `axes` object. Use this property to create a program that creates an `axes` object and plots a line connecting the locations of successive mouse clicks within the axes. Use the function `waitforbuttonpress` to wait for mouse clicks and update the plot after each click. Terminate the plot when a keyboard press occurs.

9.5 Use the MATLAB Help Browser to learn about the `CurrentCharacter` property of a `figure` object. Modify the program created in Problem 9.4 by testing the `CurrentCharacter` property when a keyboard press occurs. If the character typed on the keyboard is a "c" or "C," change the color of the line being displayed. If the character typed on the keyboard is an "s" or "S," change the line style of the line being displayed. If the character typed on the keyboard is a "w" or "W," change the width of the line being displayed. If the character typed on the keyboard is an "x" or "X," terminate the plot. (Ignore all other input characters.)

9.6 Create a MATLAB program that plots the functions

$$\begin{aligned} x(t) &= \cos\frac{t}{\pi} \\ x(t) &= 2\sin\frac{t}{2\pi} \end{aligned} \tag{9-2}$$

for the range $-2 \le t \le 2$. The program should then wait for mouse clicks, and if the mouse has clicked on one of the two lines, the program should change the line's color randomly from a choice of red, green, blue, yellow, cyan, magenta, or black. Use the function `waitforbuttonpress` to wait for mouse clicks, and update the plot after each click. Use the function `gco` to determine the object clicked on, and use the `Type` property of the object to determine if the click was on a line.

9.7 Create a MATLAB figure and store some user-defined data values in the figure. Then, save the figure in a figure file (`*.fig`) using the File/Save As menu option on the Figure Window. Next close down and restart MATLAB. Reload the figure using the File/Open menu option on a Figure Window, and try to recover the user-defined data using the `getappdata` function. Was the information preserved?

9.8 The `plot` function plots a line and returns a handle to that line. This handle can be used to get or set the line's properties after it has been created. Two of a line's properties are `XData` and `YData`, which contain the x- and y-values currently plotted. Write a program that plots the function

$$x(t) = \cos(2\pi t - \theta) \tag{9-3}$$

between the limits $-1.0 \leq t \leq 1.0$, and saves the handle of the resulting line. The angle θ is initially 0 radians. Then, re-plot line over and over with $\theta = \pi/10$ rad, $\theta = 2\pi/10$ rad, $\theta = 3\pi/10$ rad, and so forth up to $\theta = 2\pi$ rad. To re-plot the line, use a `for` loop to calculate the new values of x and t, and update the line's `XData` and `YData` properties with `set` commands. Pause 0.5 seconds between each update, using MATLAB's `pause` command.

CHAPTER 10

Graphical User Interfaces

A graphical user interface (GUI) is a pictorial interface to a program. A good GUI can make programs easier to use by providing them with a consistent appearance, and with intuitive controls such as pushbuttons, edit boxes, list boxes, sliders, and menus. The GUI should behave in an understandable and predictable manner, so that a user knows what to expect when he or she performs an action. For example, when a mouse click occurs on a pushbutton, the GUI should initiate the action described on the label of the button.

This chapter contains an introduction to the basic elements of the MATLAB GUIs. It does not contain a complete description of components or GUI features, but it does provide us with the basics required to create functional GUIs for your programs.

10.1 How a Graphical User Interface Works

A graphical user interface provides the user with a familiar environment in which to work. It contains pushbuttons, toggle buttons, lists, menus, text boxes, and so forth, all of which are already familiar to the user, so that he or she can concentrate on the purpose of the application instead of the mechanics involved in doing things. However, GUIs are harder for the programmer, because a GUI-based program must be prepared for mouse clicks (or possibly keyboard input) for any GUI element at any time. Such inputs are known as **events**, and a program that responds to events is said to be *event driven*.

The three principal elements required to create a MATLAB Graphical User Interface are:

1. **Components.** Each item on a MATLAB GUI (e.g., pushbuttons, labels, edit boxes) is a graphical component. The types of components include graphical **controls** (pushbuttons, toggle buttons, edit boxes, lists, sliders, etc.) static elements (text boxes), **menus**, **toolbars**, and **axes**. Graphical controls and text boxes are created by the function `uicontrol`, and menus are created by the functions `uimenu` and `uicontextmenu`. Toolbars are created by function `uitoolbar`. Axes, which are used to display graphical data, are created by the function `axes`.
2. **Containers.** The components of a GUI must be arranged within a **container**, which is a window on the computer screen. The most common container is a **figure**. A figure is a window on the computer screen that has a title bar along the top, and that can optionally have menus attached. In the past, figures have been created automatically whenever we plotted data. However, empty figures can be created with the function `figure`, and they can be used to hold any combination of components and other containers.

   The other types of containers are **panels** (created by the function `uipanel`) and **button groups** (created by the function `uibuttongroup`). Panels can contain components or other containers, but they do not have a title bar and cannot have menus attached. Button groups are special panels that can manage groups of radio buttons or toggle buttons to ensure that no more than one button in the group is on at any time.
3. **Callbacks.** Finally, there must be some way to perform an action if a user clicks a mouse on a button or types information on a keyboard. A mouse click or a key press is an **event**, and the MATLAB program must respond to each event if the program is to perform its function. For example, if a user clicks on a button, then that event must cause the MATLAB code that implements the function of the button to be executed. The code executed in response to an event is known as a **callback**. There must be a callback to implement the function of each graphical component on the GUI.

The basic GUI elements are summarized in Table 10.1, and some sample elements are shown in Figure 10.1 on page 442. We will study examples of these elements and then build working GUIs from them.

10.2 Creating and Displaying a Graphical User Interface

MATLAB graphical user interfaces are created using a tool called `guide`, the GUI Development Environment. This tool allows a programmer to lay out the GUI, selecting and aligning the GUI components to be placed in it. Once the components are in place, the programmer can edit their properties: name, color, size, font, text to display, and so forth. When `guide` saves the GUI, it creates a working program, including skeleton functions that the programmer can modify to implement the behavior of the GUI.

(text continues on page 443)

Table 10.1 Some Basic GUI Components

Component	Created By	Description
		Containers
Figure	`figure`	Creates a figure, which is a container that can hold components and other containers. Figures are separate windows that have title bars, and can have menus.
Panel	`uipanel`	Creates a panel, which is a container that can hold components and other containers. Unlike figures, panels do not have title bars or menus. Panels can be placed inside figures or other panels.
Button Group	`uibuttongroup`	Creates a button group, which is a special kind of panel. Button groups automatically manage groups of radio buttons or toggle buttons to ensure that only one item of the group is on at any given time.
		Graphical Controls
Pushbutton	`uicontrol`	A graphical component that implements a pushbutton. It triggers a callback when clicked with a mouse.
Toggle Button	`uicontrol`	A graphical component that implements a toggle button. A toggle button is either "on" or "off," and it changes state each time that it is clicked. Each mouse button click also triggers a callback.
Radio Button	`uicontrol`	A radio button is a type of toggle button that appears as a small circle with a dot in the middle when it is "on." Groups of radio buttons are used to implement mutually exclusive choices. Each mouse click on a radio button triggers a callback.
Check Box	`uicontrol`	A checkbox is a type of toggle button that appears as a small square with a check mark in it when it is "on." Each mouse click on a check box triggers a callback.
Edit Box	`uicontrol`	An edit box displays a text string and allows the user to modify the information displayed. A callback is triggered when the user presses the `Enter` key, or when the user clicks in a different object with the mouse.
List Box	`uicontrol`	A list box is a graphical control that displays a series of text strings. A user may select one of the text strings by single- or double-clicking on them. A callback is triggered when the user selects a string.
Popup Menus	`uicontrol`	A popup menu is a graphical control that displays a series of text strings in response to a mouse click. When the popup menu is not clicked on, only the currently selected string is visible.
Slider	`uicontrol`	A slider is a graphical control to adjust a value in a smooth, continuous fashion by dragging the control with a mouse. Each slider change triggers a callback.

(continued)

Table 10.1 (continued)

Component	Created By	Description
		Static Elements
Frame	uicontrol	Creates a frame, which is a rectangular box within a figure. Frames are used to group sets of controls together. Frames never trigger callbacks. (This is a deprecated component, which should not be used in new GUIs.)
Text Field	uicontrol	Creates a label, which is a text string located at a point on the figure. Text fields never trigger callbacks.
		Menus, Toolbars, Axes
Menu Items	uimenu	Creates a menu item. Menu items trigger a callback when a mouse button is released over them.
Context Menus	uicontextmenu	Creates a context menu, which is a menu that appears over a graphical object when a user right-clicks the mouse on that object.
Toolbar	uitoolbar	Creates a toolbar, which is a bar across the top of the figure containing quick-access buttons.
Toolbar Pushbutton	uipushtool	Creates a pushbutton to go in a toolbar.
Toolbar Toggle	uitoggletool	Creates a toggle button to go in a toolbar.
Button	axes	Creates a new set of axes to display data on. Axes never trigger callbacks.
Axes		

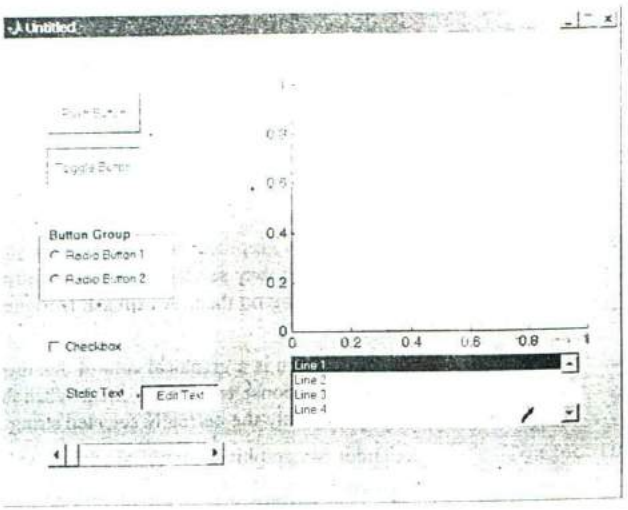


Figure 10.1 A Figure Window showing examples of MATLAB GUI elements. From top to bottom and left to right, the elements are: (1) a pushbutton; (2) a toggle button in the 'on' state; (3) two radio buttons within a button group; (4) a check box; (5) a label and an edit box; (6) a slider; (7) a set of axes; and (8) a list box.

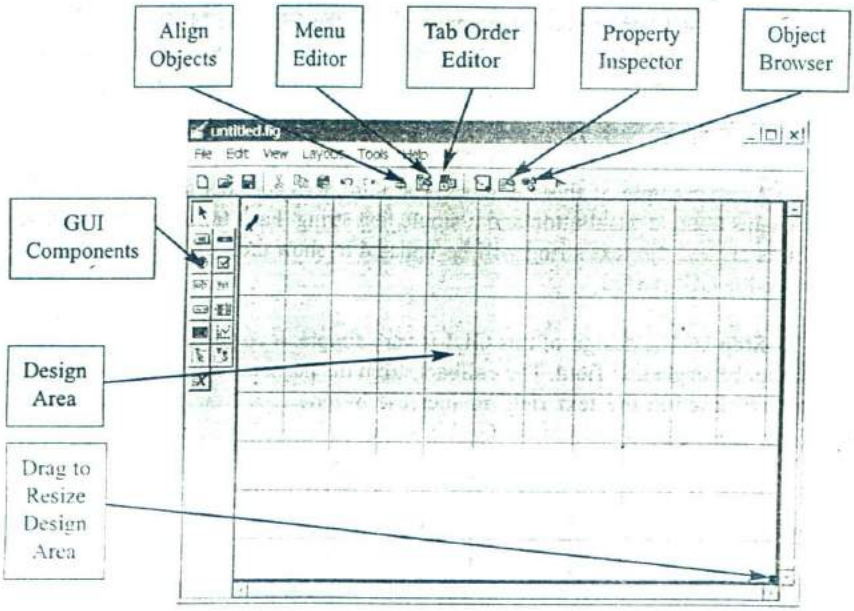


Figure 10.2 The `guide` tool window.

When `guide` is executed, it creates the Layout Editor, shown in Figure 10.2. The large gray area with grid lines is the *layout area*, where a programmer can lay out the GUI. The Layout Editor window has a palette of GUI components along the left-hand side of the layout area. A user can create any number of GUI components by first clicking on the desired component and then dragging its outline in the layout area. The top of the window has a toolbar with a series of useful tools that allow the user to distribute and align GUI components, modify the properties of GUI components, and add menus to GUIs, among other things.

The basic steps required to create a MATLAB GUI are:

1. Decide what elements are required for the GUI and what the function of each element will be. Make a rough layout of the components by hand on a piece of paper.
2. Use the MATLAB tool called `guide` (GUI Development Environment) to lay out the components on a figure. The size of the figure as well as the alignment and spacing of components on the figure can be adjusted using the tools built into `guide`.
3. Use a MATLAB tool called the Property Inspector (built into `guide`) to give each component a name (a "tag") and to set the characteristics of each component, such as its color or the text it displays.
4. Save the figure to a file. When the figure is saved, two files will be created on disk with the same name but different extents. The `fig` file contains

the GUI layout and the components of the GUI; the M-file contains the code to load the figure along with skeleton callback functions for each GUI element.

5. Write code to implement the behavior associated with each callback function.

As an example of this stepwise procedure, let's consider a simple GUI that contains a single pushbutton and a single text string. Each time that the pushbutton is clicked, the text string will be updated to show the total number of clicks since the GUI started.

Step 1: The design of this GUI is very simple. It contains a single pushbutton and a single text field. The callback from the pushbutton will cause the number displayed in the text field to increase by one each time that the button is pressed. A rough sketch of the GUI is shown in Figure 10.3.

Step 2: To lay out the components on the GUI, run the MATLAB function `guide`. When `guide` is executed, it creates the window shown in Figure 10.2.

First, we must set the size of the layout area, which will become the size of the final GUI. We do this by dragging the small square on the lower right-hand corner of the layout area until it has the desired size and shape. Then, click on the "pushbutton" button in the list of GUI components and create the shape of the pushbutton in the layout area. Finally, click on the "text" button in the list of GUI components and create the shape of the text field in the layout area. The resulting figure after these steps is shown in Figure 10.4. We could now adjust the alignment of these two elements using the Alignment Tool, if desired.

Step 3: To set the properties of the pushbutton, click on the button in the layout area and then select "Property Inspector" ( ) from the toolbar. Alternatively, right-click on the button and select "Property Inspector" from the popup menu. The Property Inspector window shown in Figure 10.5 will appear. Note that this window lists every property available for the pushbutton and allows us to set each value using a GUI interface. The Property Inspector performs the

(text continues on page 446)

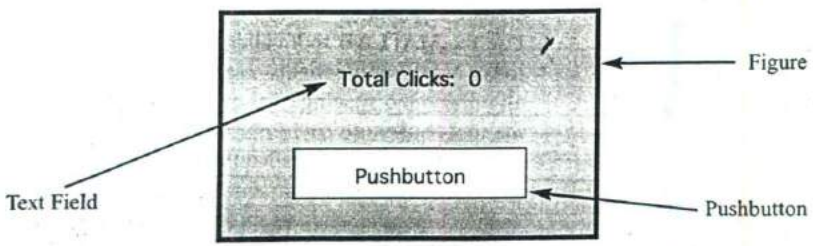


Figure 10.3 Rough layout for a GUI containing a single pushbutton and a single label field.

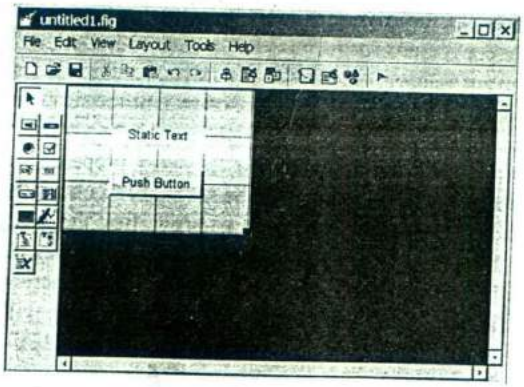


Figure 10.4 The completed GUI layout within the `guide` window.

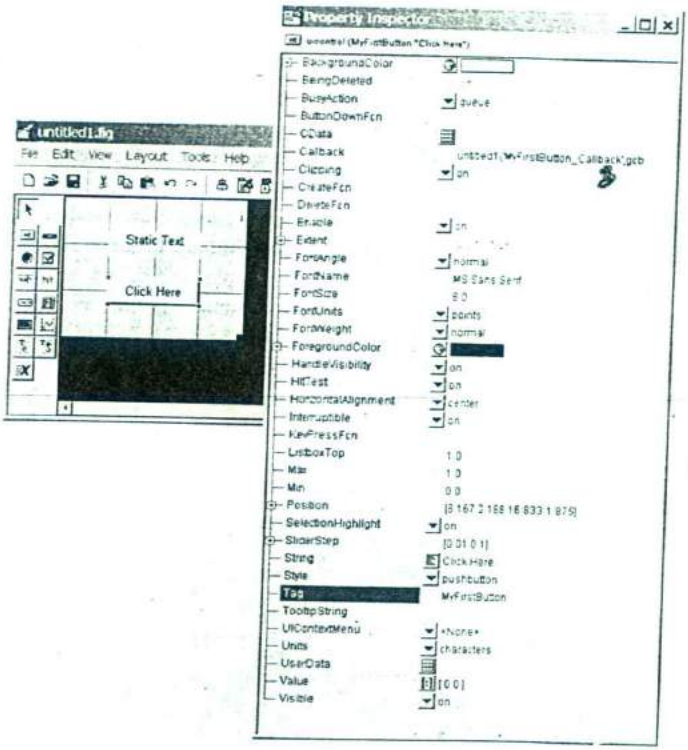


Figure 10.5 The Property Inspector showing the properties of the pushbutton. Note that the String is set to `'Click Here'`, and the Tag is set to `'MyFirstButton'`.

same function as the `get` and `set` functions introduced in Chapter 9, but in a much more convenient form.

For the pushbutton, we may set many properties such as color, size, font, and text alignment. However, we *must* set two properties: the `String` property, which contains the text to be displayed, and the `Tag` property, which is the name of the pushbutton. In this case, the `String` property will be set to `'Click Here'`, and the `Tag` property will be set to `MyFirstButton`.

For the text field, we *must* set two properties: the `String` property, which contains the text to be displayed, and the `Tag` property, which is the name of the text field. This name will be needed by the callback function to locate and update the text field. In this case, the `String` property will be set to `'Total Clicks: 0'`, and the `Tag` property defaulted to `'MyFirstText'`. The layout area after these steps is shown in Figure 10.6.

It is possible to set the properties of the figure itself by clicking on a clear spot in the Layout Editor, and using the Property Inspector to examine and set the figure's properties. Although not required, it is a good idea to set the figure's `Name` property. The string in the `Name` property will be displayed in the title bar of the resulting GUI when it is executed. In this program, we will set the `Name` to `'MyFirstGUI'`.

Step 4: We will now save the layout area under the name `MyFirstGUI`. Select the "File/Save As" menu item, type the name `MyFirstGUI` as the file name, and click "Save". This action will automatically create two files—`MyFirstGUI.fig` and `MyFirstGUI.m`. The figure file contains the actual GUI that we have created. The M-file contains code that loads the figure file and creates the GUI, plus a skeleton callback function for each active GUI component.

At this point, we have a complete GUI, but one that does not yet do the job it was designed to do. You can start this GUI by typing `MyFirstGUI` in the Command Window, as shown in Figure 10.7. If the button is clicked on this GUI, nothing happens.

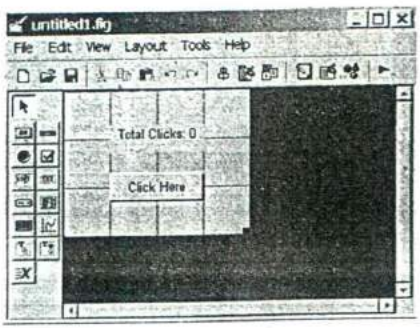


Figure 10.6 The design area after the properties of the pushbutton and the text field have been modified.

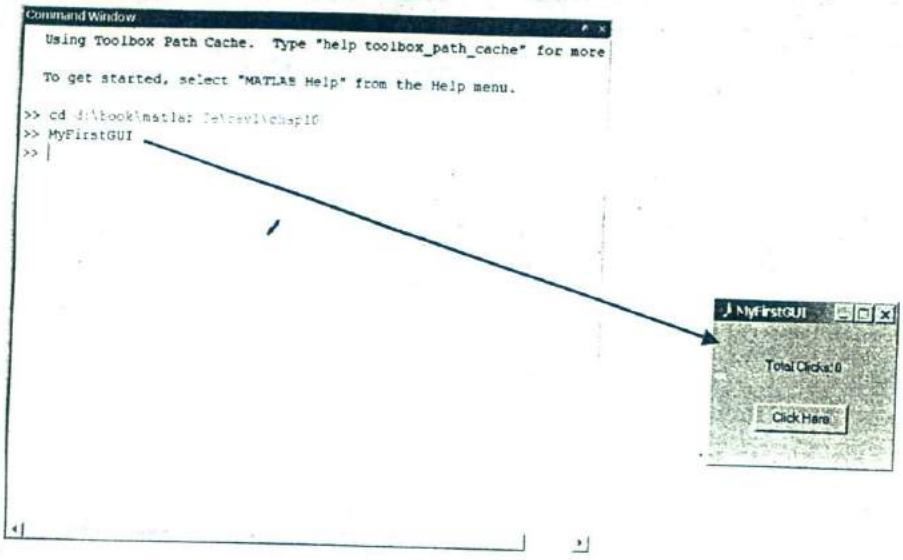


Figure 10.7 Typing `MyFirstGUI` in the Command Window starts the GUI.

A portion of the M-file automatically created by `guide` is shown in Figure 10.8. This file contains the main function `MyFirstGUI`, plus subfunctions to specify the behavior of the active GUI components. The file contains a *dummy callback function for every active GUI component* that you defined. In this case, the only active GUI component was the pushbutton, so there is a callback function called `MyFirstButton_Callback`, which is executed when the user clicks on the button.

If function `MyFirstGUI` is called *without* arguments, then the function displays the GUI contained in file `MyFirstGUI.fig`. If function `MyFirstGUI` is called *with* arguments, then the function assumes that the first argument is the name of a subfunction, and it calls that subfunction using `feval`, passing the other arguments on to that subfunction.

Each callback function handles events from a single GUI component. If a mouse click (or keyboard input for edit fields) occurs on the GUI component, then the component's callback function will be automatically called by MATLAB. The name of the callback function will be the value in the `Tag` property of the GUI component plus the characters "`_Callback`." Thus, the callback function for `MyFirstButton` will be named `MyFirstButton_Callback`.

M-files created by `guide` contain callbacks for each active GUI component, but these callbacks don't do anything yet.

Step 5: Now, we need to write the callback subfunction code for the pushbutton. This function will include a `persistent` variable that can be used

```
function varargout = MyFirstGUI(varargin)
% MYFIRSTGUI M-file for MyFirstGUI.fig
%      MYFIRSTGUI, by itself, creates a new MYFIRSTGUI or raises the existing
%      singleton*.
%
%      H = MYFIRSTGUI returns the handle to a new MYFIRSTGUI or the handle to
%      the existing singleton*.
%
%      MYFIRSTGUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in MYFIRSTGUI.M with the given input arguments.
%
%      MYFIRSTGUI('Property','Value',...) creates a new MYFIRSTGUI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before MyFirstGUI_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to MyFirstGUI_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help MyFirstGUI

% Last Modified by GUIDE v2.5 11-Feb-2004 16:17:45

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @MyFirstGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @MyFirstGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before MyFirstGUI is made visible.
function MyFirstGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to MyFirstGUI (see VARARGIN)

% Choose default command line output for MyFirstGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

Main Function

Figure Opening Function

Figure 10.8 The M-file for `MyFirstGUI`, automatically created by `guide`.

```
% UIWAIT makes MyFirstGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = MyFirstGUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in MyFirstButton.
function MyFirstButton_Callback(hObject, eventdata, handles)
% hObject    handle to MyFirstButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Data Output Function

Button Callback Function

Figure 10.8 *(continued)*

to count the number of clicks that have occurred. When a click occurs on the pushbutton, MATLAB will call the function `MyFirstGUI` with `MyFirstButton_Callback` as the first argument. Then function `MyFirstGUI` will call subfunction `MyFirstButton_Callback`, as shown in Figure 10.9. This function should increase the count of clicks by one, create a new text string containing the count, and store the new string in the `String` property of the text field `MyFirstText`. A function to perform this step is shown below:

```
function MyFirstButton_Callback(hObject, eventdata, handles)
% hObject    handle to MyFirstButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Declare and initialize variable to store the count
persistent count
if isempty(count)
   count = 0;
end

% Update count
count = count + 1;

% Create new string
str = sprintf('Total Clicks: %d',count);

% Update the text field
set (handles.MyFirstText,'String',str);
```

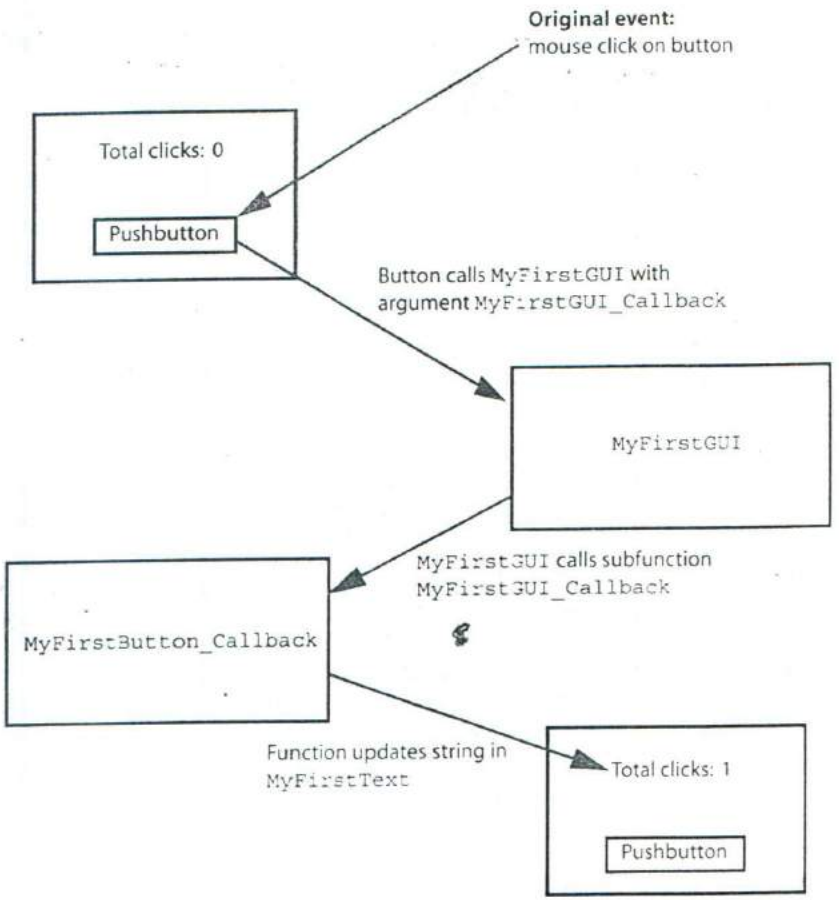


Figure 10.9 Event handling in program `MyFirstGUI`. When a user clicks on the button with the mouse, the function `MyFirstGUI` is called automatically with the argument `MyFirstButton_Callback`. Function `MyFirstGUI` in turn calls subfunction `MyFirstButton_Callback`. This function increments `count`, and then saves the new count in the text field on the GUI.

Note that this function declares a persistent variable `count` and initializes it to zero. Each time the function is called, it increments `count` by 1 and creates a new string containing the count. Then, it updates the string displayed in the text field `MyFirstText`.

The resulting program is executed by typing `MyFirstGUI` in the Command Window. When the user clicks on the button, MATLAB automatically calls function `MyFirstGUI` with `MyFirstButton_Callback` as the first argument,



Figure 10.10 The resulting program after three button pushes.

and function `MyFirstGUI` calls subfunction `MyFirstButton_Callback`. This function increments variable `count` by one and updates the value displayed in the text field. The resulting GUI after three button pushes is shown in Figure 10.10.

Good Programming Practice

Use `guide` to lay out a new GUI, and use the Property Inspector to set the initial properties of each component, such as the text displayed on the component, the color of the component, and the name of the callback function, if required.

Good Programming Practice

After creating a GUI with `guide`, manually edit the resulting M-file to add comments describing its purpose and components, and to implement the behavior of callbacks.

A Look Under the Hood

Figure 10.8 shows the M-file that was automatically generated by `guide` for `MyFirstGUI`. We will now examine this M-file more closely to understand how it works.

First, let's look at the main `function` declaration itself. Note that this function uses `varargin` to represent its input arguments and `varargout` to represent its output results. As we learned in Chapter 7, function `varargin` can represent an arbitrary number of input arguments, and function `varargout` can represent a varying number of output arguments. Therefore, *a user can call function* `MyFirstGUI` *with any number of arguments.*

The main function begins with a series of comments that serve as the help message displayed when the user types "`help MyFirstGUI`." You should edit these comments to reflect the actual function of your program.

Next, the main function creates a structure called `gui_State`. The code to create this structure is shown below:

```
gui_Singleton = 1;
gui_State = struct('gui_Name',          mfilename, ...
                   'gui_Singleton',     gui_Singleton, ...
                   'gui_OpeningFcn',    @MyFirstGUI_OpeningFcn, ...
                   'gui_OutputFcn',     @MyFirstGUI_OutputFcn, ...
                   'gui_LayoutFcn',     [] , ...
                   'gui_Callback',      []);
if nargin & isstr(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end
```

The structure contains some control information, plus function handles for some of the subfunctions in the file. Other MATLAB GUI functions use these function handles to call the subfunctions from outside of the M-file. Note that the first argument is converted into a callback function handle, if it exists.

The value `gui_Singleton` specifies whether there can be one or more simultaneous copies of the GUI. If `gui_Singleton` is 1, then there can be only one copy of the GUI. If `gui_Singleton` is 0, then there can be many simultaneous copies of the GUI.

The main function calls the MATLAB function `gui_mainfcn`, and passes the `gui_State` structure and all of the input arguments to it. Function `gui_mainfcn` is standard MATLAB function. It actually does the work of creating the GUI, or of calling the appropriate subfunction in response to a callback.

Calling the M-File Without Arguments

If the user calls `MyFirstGUI` *without* arguments, function `gui_mainfcn` loads the GUI from the figure file `MyFirstGUI.fig` using the `openfig` function. The form of this function is

```
fig = openfig(mfilename,reuse);
```

where `mfilename` is the name of the figure file to load. The second argument in the function specifies whether there can be only one copy of the figure running at a given time, or multiple copies can be run. If `gui_State.gui_Singleton` is 1, then the second argument is set to `'reuse'`, and only one copy of the figure can be run. If `openfig` is called with the `'reuse'` option and the specified figure already exists, the preexisting figure will be brought to the top of the screen and reused. In contrast, if `gui_State.gui_Singleton` is 0, then the argument is set to `'new'` and a new copy of the figure will be created each time that `MyFirstGUI` is called without arguments. By default, a GUI

created by `guide` has the `gui_State.gui_Singleton` set to 1, so only one copy of the figure can exist at any time. If you wish to have multiple copies of the GUI, turn off the "GUI allows only one instance to run" flag in the GUI Options selection on `guide`'s Tools menu.

After the figure is loaded, the `gui_mainfcn` function executes the statement

```
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
```

This function sets the background color of the figure to match the default background color used by the computer that MATLAB is executing on. It makes the color of the GUI match the color of native windows on the computer. Therefore, a GUI can be written on a Windows-based PC and used on a Unix-based computer, and vice versa. It will look natural in either environment.

Then, function `gui_mainfcn` creates a structure containing the handles of all the objects in the current figure and stores that structure as application data in the figure.

```
guidata(gui_hFigure, guihandles(gui_hFigure));
```

Function `guihandles` creates a structure containing handles to all of the objects within the specified figure. The element names in the structure correspond to the `Tag` properties of each GUI component, and the values are the handles of each component. For example, the handle structure returned in `MyFirstGUI.m` is

```
» handles = guihandles(fig)
handles =
          figure1: 99.0005
     MyFirstText: 3.0021
   MyFirstButton: 100.0007
```

There are three GUI components in this figure—the figure itself, plus a text field and a pushbutton. Function `guidata` saves the handles structure as application data in the figure, using the `setappdata` function that we studied in Chapter 9.

Finally, just before making the figure visible, function `gui_mainfcn` calls the function specified in `gui_OpeningFcn`. This function provides a way for the programmer to customize the GUI before showing it to the user. For example, a programmer could load initial data values here, change background colors, and so forth.

Calling the M-File with Arguments

When the user clicks on an active GUI element, MATLAB calls `MyFirstGUI` with the name of the GUI element's callback function in the first argument. If `MyFirstGUI` is called *with* arguments, the value returned by `nargin` will be greater than zero. In this case, function `MyFirstGUI` converts the callback function name into a function handle using the following code.

```
if nargin & isstr(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end
```

When function `gui_mainfcn` is called this time, it calls the callback function using this function handle. The callback executes and responds to the mouse click or keyboard input, as appropriate.

Figure 10.11 summarizes the operation of `MyFirstGUI` on first and subsequent calls.

The Structure of a Callback Subfunction

Every callback subfunction has the standard form

```
function ComponentTag_Callback(hObject, eventdata, handles)
```

where `ComponentTag` is the name of the component generating the callback (the string in its `Tag` property). The arguments of this subfunction are:

- **`hObject`**—The handle of the parent figure
- **`eventdata`**—A currently unused (in MATLAB 7) array.
- **`Handles`**—The handles structure contains the handles of all GUI components on the figure.

Note that each callback function has full access to the `handles` structure; therefore, each callback function can modify any GUI component in the figure. We took advantage of this structure in the callback function for the pushbutton in `MyFirstGUI`, where the callback function for the pushbutton modified the text displayed in the text field.

```
% Update the text field
set (handles.MyFirstText,'String',str);
```

Adding Appication Data to a Figure

It is possible to store any application-specific information needed by a GUI program in the `handles` structure instead of using global or persistent memory for that data. The resulting GUI design is more robust, since other MATLAB programs cannot accidentally modify the global GUI data and since multiple copies of the same GUI cannot interfere with each other.

To add local data to the `handles` structure, we must manually modify the M-file after it is created by `guide`. A programmer first adds the required local data to the `handles` structure, and then calls `guidata` to update the `handles` structure stored in the figure. For example, to add the number of mouse clicks `count` to the `handles` structure, we would modify the `MyFirstButton_Callback` function as follows:

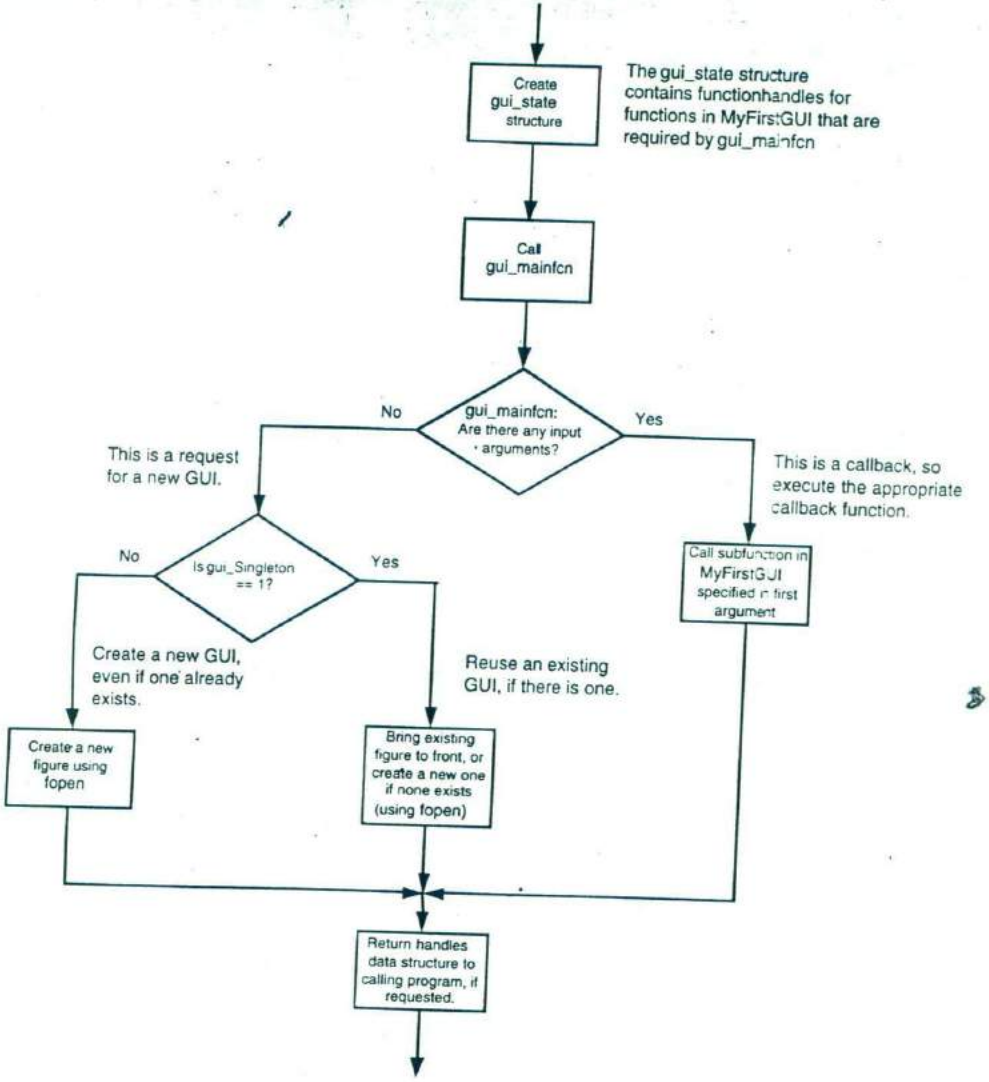


Figure 10.11 The operation of `MyFirstGUI`. If there are no calling arguments, it either creates a GUI or displays an existing GUI. If there are calling arguments, the first arguments is assumed to be a callback function name, and `MyFirstGUI` calls the appropriate callback function.

```
function MyFirstButton_Callback(hObject, eventdata, handles)
% hObject    handle to MyFirstButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Create the count field if it does not exist
if ~isfield(handles,'count')
   handles.count = 0;
end

% Update count
handles.count = handles.count + 1;

% Save the updated handles structure
guidata(hObject, handles);

% Create new string
str = sprintf('Total Clicks: %d',handles.count);

% Update the text field
set (handles.MyFirstText,'String',str);
```

Good Programming Practice

Store GUI application data in the `handles` structure, so that it will automatically be available to any callback function.

Good Programming Practice

If you modify any of the GUI application data in the `handles` structure, be sure to save the structure with a call to `guidata` before exiting the function where the modifications occurred.

A Few Useful Functions

Three special functions are occasionally used in the design of callback functions: `gcbo`, `gcbf`, and `findobj`. These functions are not actually needed with MATLAB 7 GUIs, because the same information is available in the `handles` data structure. However, they were commonly used in earlier versions of MATLAB, and a programmer is sure to encounter them.

Function `gcbo` (*get callback object*) returns the handle of the object that generated the callback, while function `gcbf` (*get callback figure*) returns the handle of the figure containing that object. These functions can be used by a callback function to determine the object and figure producing the callback so that it can modify objects on that figure.

Function `findobj` searches through all of the child objects within a parent object, looking for ones have a specific value of a specified property. It returns a handle to any objects with the matching characteristics. The most common form of `findobj` is

```
Hndl = findobj(parent,'Property',Value);
```

where `parent` is the handle of a parent object such as a figure, `'Property'` is the property to examine, and `'Value'` is the `value` to look for.

For example, suppose that a programmer would like to change the color of all the lines in a plot on the callback figure. He or she could find the lines and change the line colors to red with the following statements:

```
Hndl = findobj(gcbf, 'Type', 'Line');
for ii = 1:length(Hndl)
   set( Hndl, 'Color','r' );
end
```

10.3 Object Properties

Every GUI object includes an extensive list of properties that can be used to customize the object. These properties are slightly different for each type of object (figures, axes, `uicontrols`, etc.). All of the properties for all types of objects are documented on the on-line Help Browser, but a few of the more important properties for `figure` and `uicontrol` objects are summarized in Tables 10.2 and 10.3.

Table 10.2 Important `figure` Properties

Property	Description
`Color`	Specifies the color of the figure. The value is either a predefined color such as `'r'`, `'g'`, or `'b'`, or else a 3-element vector specifying the red, green, and blue components of the color on a 0–1 scale. For example, the color magenta would be specified by `[1 0 1]`.
`CurrentCharacter`	Contains the character corresponding to the last key pressed in this figure.
`CurrentPoint`	Location of the last button click in this figure, measured from the lower left-hand corner of the figure in units specified in the `Units` property.
`Dockable`	Specifies whether or not the figure can be docked to the desktop. Possible values are `'on'` or `'off'`.
`MenuBar`	Specifies whether or not the default set of menus appears on the figure. Possible values are `'figure'` to display the default menus or `'none'` to delete them.
`Name`	A string containing the name that appears in the title bar of a figure.
`NumberTitle`	Specifies whether or not the figure number appears in the title bar. Possible values are `'on'` or `'off'`.

(continued)

Table 10.2 (continued)

Property	Description
`Position`	Specifies the position of a figure on the screen, in the units specified by the `'units'` property. This value accepts a 4-element vector in which the first two elements are the *x* and *y* positions of the lower left-hand corner of the figure and the next two elements are the width and height of the figure.
`SelectionType`	Specifies the type of selection for the last mouse click on this figure. A single click returns type `'normal'`, while a double click returns type `'open'`. There are additional options; see the MATLAB on-line documentation.
`Tag`	The "name" of the figure, which can be used to locate it.
`Units`	The units used to describe the position of the figure. Possible choices are `'inches'`, `'centimeters'`, `'normalized'`, `'points'`, `'pixels'`, or `'characters'`. The default units are `'pixels'`.
`Visible`	Specifies whether or not this figure is visible. Possible values are `'on'` or `'off'`.
`WindowStyle`	Specifies whether this figure is normal or modal (see discussion of Dialog Boxes). Possible values are `'normal'` or `'modal'`.

Table 10.3 Important `uicontrol` Properties

Property	Description
`BackgroundColor`	Specifies the background color of the object. The value is either a predefined color such as `'r'`, `'g'`, or `'b'`, or else a 3-element vector specifying the red, green, and blue components of the color on a 0–1 scale. For example, the color magenta would be specified by `[1 0 1]`.
`Callback`	Specifies the name and parameters of the function to be called when the object is activated by a keyboard or text input.
`Enable`	Specifies whether or not this object is selectable. If it is not enabled, it will not respond to mouse or keyboard input. Possible values are `'on'` or `'off'`.
`FontAngle`	A string containing the font angle for text displayed on the object. Possible values are `'normal'`, `'italic'`, and `'oblique'`.
`FontName`	A string containing the font name for text displayed on the object.
`FontSize`	A number specifying the font size for text displayed on the object.
`FontUnits`	The units in which the font size is defined. Possible choices are `'inches'`, `'centimeters'`, `'normalized'`, `'points'`, and `'pixels'`. The default font units are `'points'`.
`FontWeight`	A string containing the font weight for text displayed on the object. Possible values are `'light'`, `'normal'`, `'demi'`, and `'bold'`. The default font weight is `'normal'`.
`ForegroundColor`	Specifies the foreground color of the object.

(continued)

Table 10.3 (continued)

Property	Description
`HorizontalAlignment`	Specifies the horizontal alignment of a text string within the object. Possible values are `'left'`, `'center'`, and `'right'`.
`Max`	The maximum size of the `value` property for this object.
`Min`	The minimum size of the `value` property for this object.
`Parent`	The handle of the figure containing this object.
`Position`	Specifies the position of the object on the screen, in the units specified by the `'units'` property. This value accepts a 4-element vector in which the first two elements are the *x* and *y* positions of the lower left-hand corner of the object *relative the figure containing it*, and the next two elements are the width and height of the object.
`Tag`	The "name" of the object, which can be used to locate it.
`TooltipString`	Specifies the help text to be displayed when a user places the mouse pointer over an object.
`Units`	The units used to describe the position of the figure. Possible choices are `'inches'`, `'centimeters'`, `'normalized'`, `'points'`, `'pixels'`, or `'characters'`. The default units are `'pixels'`.
`Value`	The current value of the `uicontrol`. For toggle buttons, check boxes, and radio buttons, the value is `max` when the button is on and `min` when the button is off. Other controls have different meanings for this term.
`Visible`	Specifies whether or not this object is visible. Possible values are `'on'` or `'off'`.

Object properties can be modified using either the Property Inspector or the `get` and `set` functions. Although the Property Inspector is a convenient way to adjust properties during GUI design, we must use `get` and `set` to adjust them dynamically from within a program, such as in a callback function.

10.4 Graphical User Interface Components

This section summarizes the basic characteristics of common graphical user interface components. It describes how to create and use each component, as well as the types of events each component can generate. The components discussed in this section are:

- Static Text Fields
- Edit Boxes
- Pushbuttons

- Toggle Buttons
- Checkboxes
- Radio Buttons
- Popup Menus
- List Boxes
- Sliders

Static Text Fields

A **static text field** is a graphical object that displays one or more text strings, which are specified in the text field's `String` property. The `String` property accepts a string or a cell array of strings. If the input value is a string, it will be displayed on a single line. If the input value is a cell array of strings, the first element will be displayed on the first line of the text box, the second element will be displayed on the second line of the text box, and so forth. You can specify how the text is aligned in the display area by setting the horizontal alignment property. By default, text fields are horizontally centered. A text field is created by a `uicontrol` whose style property is `'text'`. A text field may be added to a GUI by using the text tool ( ) in the Layout Editor.

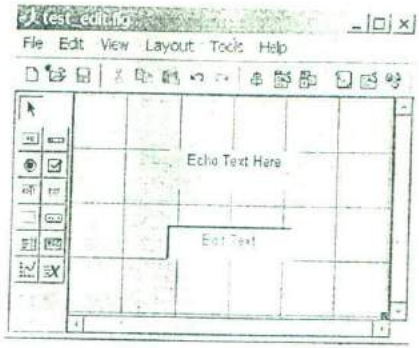
Text fields do not create callbacks, but the value displayed in the text field can be updated from another component's callback function by changing the text field's `String` property, as shown in program `MyFirstGUI` in Section 10.2.

Edit Boxes

An **edit box** is a graphical object that allows a user to enter one or more text strings. It is created by a `uicontrol` whose style property is `'edit'`. If the `min` property and `max` property are both set to 1, then *the edit box will accept a single line of text*, and it will generate a callback when the user presses the Enter key or the ESC key after typing the text.

Figure 10.12*a* shows a simple GUI containing an edit box named `'EditBox'` and a text field named `'TextBox'`. When a user presses Enter or ESC after typing a string into the edit box, the program automatically calls the function `EditBox_Callback`, which is shown in Figure 10.12*b*. This function locates the edit box using the `handles` structure, and recovers the string typed by the user. Then, it locates the text field and displays the string in the text field. Figure 10.13 shows this GUI just after it has started, and after the user has typed the word "Hello" in the edit box.

If the `max` property is set to a number greater than the `min` property, *the edit box will accept as many lines of text as the user wishes to enter*. The textbox will include a vertical scrollbar to allow the user to move up and down through the data. Either the scrollbar or the up and down arrows can be used to move between the input lines. If the user presses the Enter key in a multi-line edit box, the current line is finished and the cursor moves down to the next line for additional



(a)

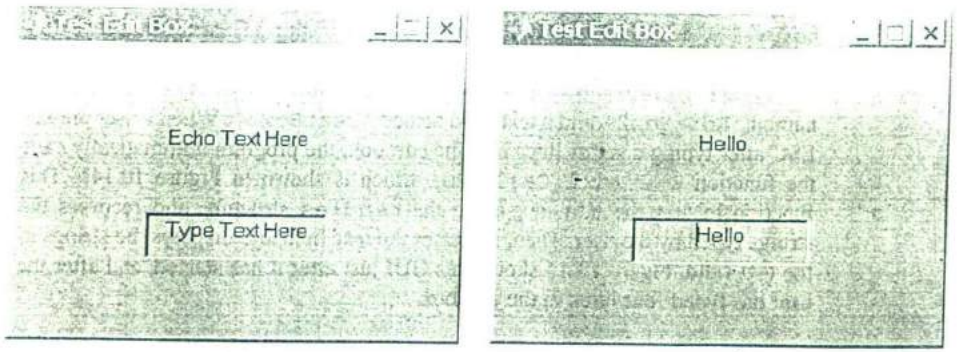
```
function EditBox_Callback(hObject, eventdata, handles

% Find the value typed into the edit box
str = get (handles.EditBox,'String');

% Place the value into the text field
set (handles.TextBox,'String',str);
```

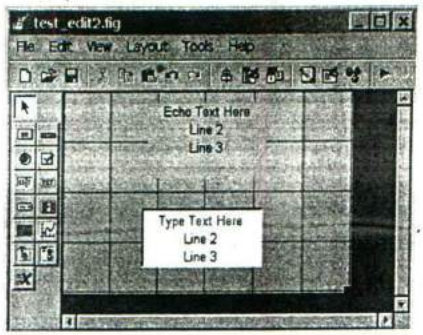
(b)

Figure 10.12 *(a)* Layout of a simple GUI with a single-line edit box and a text field. *(b)* The callback function for this GUI.



(a) *(b)*

Figure 10.13 *(a)* The GUI produced by program `test_edit`. *(b)* The GUI after a user types `'Hello'` into the edit box and presses Enter.



(a)

```
function EditBox2_Callback(hObject, eventdata, handles)

% Find the value typed into the edit box
str = get (handles.EditBox,'String');

% Place the value into the text field
set (handles.TextBox2,'String',str);
```

(b)

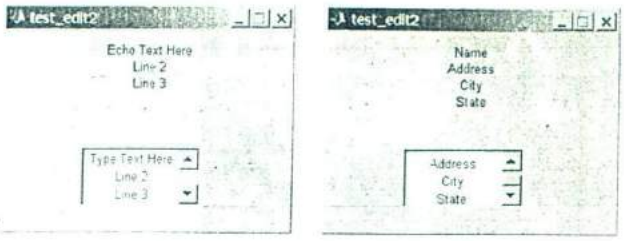
Figure 10.14 *(a)* Layout of a simple GUI with a multi-line edit box and a text field. *(b)* The callback function for this GUI.

input. If the user presses the ESC key or clicks a point on the figure background with the mouse, a callback will be generated and the data typed into the edit box will be available as a cell array of strings in the `uicontrol`'s `String` property.

Figure 10.14*a* shows a simple GUI containing a multi-line edit box named `'EditBox2'` and a text field named `'TextBox2'`. When a user presses ESC after typing a set of lines into the edit box, the program automatically calls the function `EditBox2_Callback`, which is shown in Figure 10.14*b*. This function locates the edit box using the `handles` structure and recovers the strings typed by the user. Then, it locates the text field and displays the strings in the text field. Figure 10.15 shows this GUI just after it has started, and after the user has typed four lines in the edit box.

Pushbuttons

A **pushbutton** is a component that a user can click on to trigger a specific action. The pushbutton generates a callback when the user clicks on it with the mouse. A pushbutton is created by creating a `uicontrol` whose style property



(a) *(b)*

Figure 10.15 *(a)* The GUI produced by program `test_edit2`. *(b)* The GUI after a user types four lines into the edit box and presses ESC.

is `'pushbutton'`. It can be added to a GUI by using the pushbutton tool ( ) in the Layout Editor.

Function `MyFirstGUI` in Figure 10.10 illustrates the use of pushbuttons.

Toggle Buttons

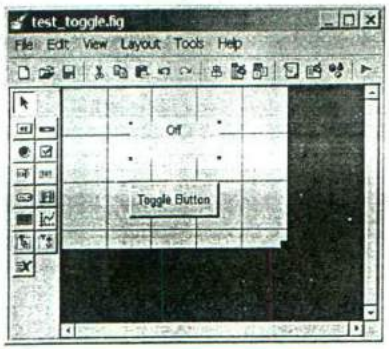
A **toggle button** is a type of button that has two states: on (depressed) and off (not depressed). A toggle button switches between these two states whenever the mouse clicks on it, and it generates a callback each time. The `'Value'` property of the toggle button is set to `max` (usually 1) when the button is on, and `min` (usually 0) when the button is off.

A toggle button is created by a `uicontrol` whose style property is `'togglebutton'`. It can be added to a GUI by using the toggle button tool ( ) in the Layout Editor.

Figure 10.16*a* shows a simple GUI containing a toggle button named `'ToggleButton'` and a text field named `'TextBox'`. When a user clicks on the toggle button, it automatically calls the function `ToggleButton_Callback`, which is shown in Figure 10.16*b*. This function locates the toggle button using the `handles` structure and recovers its state from the `'Value'` property. Then, it locates the text field and displays the state in the text field. Figure 10.17 shows this GUI just after it has started, and after the user has clicked on the toggle button for the first time.

Checkboxes and Radio Buttons

Checkboxes and radio buttons are essentially identical to toggle buttons except that they have different shapes. Like toggle buttons, they have two states: on and off. They switch between these two states whenever the mouse clicks on them,



(a)

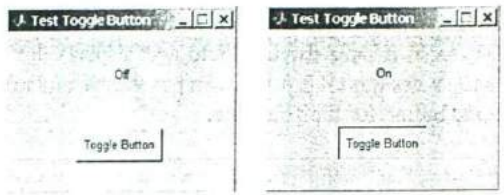
```
function ToggleButton_Callback(hObject, eventdata, handles)
% hObject    handle to ToggleButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Find the state of the toggle button
state = get(handles.ToggleButton,'Value');

% Place the value into the text field
if state == 0
   set (handles.TextBox,'String','Off');
else
   set (handles.TextBox,'String','On');
end
```

(b)

Figure 10.16 *(a)* Layout of a simple GUI with a toggle button and a text field. *(b)* The callback function for this GUI.



(a) *(b)*

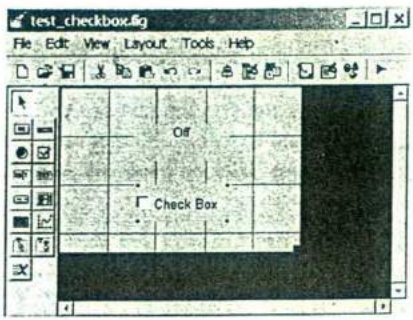
Figure 10.17 *(a)* The GUI produced by program `test_togglebutton` when the toggle button is off. *(b)* The GUI when the toggle button is on.

generating a callback each time. The `'Value'` property of the checkbox or radio button is set to max (usually 1) when they are on, and min (usually 0) when they are off. Both checkboxes and radio buttons are illustrated in Figure 10.1.

A checkbox is created by a `uicontrol` whose style property is `'checkbox'`, and a radio button is created by a `uicontrol` whose style property is `'radiobutton'`. A checkbox may be added to a GUI by using the checkbox tool (☑) in the Layout Editor, and a radio button may be added to a GUI by using the radio button tool (◉) in the Layout Editor.

Checkboxes are traditionally used to display on/off options, and groups of radio buttons are traditionally used to select among mutually exclusive options.

Figure 10.18*a* shows shows a simple GUI containing a checkbox named `'CheckBox'` and a text field named `'TextBox'`. When a user clicks on the checkbox, it automatically calls the function `CheckButton_Callback`, which is shown in Figure 10.18*b*. This function locates the checkbox using the `handles` structure and recovers its state from the `'Value'` property. Then, it



(a)

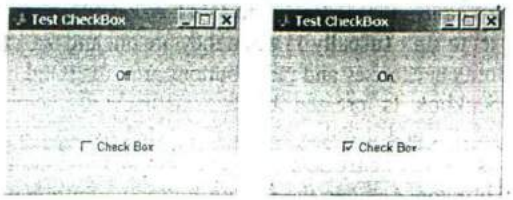
```
function CheckBox_Callback(hObject, eventdata, handles)

% Find the state of the checkbox
state = get(handles.CheckBox,'Value');

% Place the value into the text field
if state == 0
   set (handles.TextBox,'String','Off');
else
   set (handles.TextBox,'String','On');
end ;
```

(b)

Figure 10.18 *(a)* Layout of a simple GUI with a CheckBox and a text field. *(b)* The callback function for this GUI.



(a) (b)

Figure 10.19 *(a)* The GUI produced by program `test_checkbox` when the toggle button is off. *(b)* The GUI when the toggle button is on.

locates the text field and displays the state in the text field. Figure 10.19 shows this GUI just after it has started, and after the user has clicked on the toggle button for the first time.

Figure 10.20*a* shows an example of how to create a group of mutually exclusive options with radio buttons. The GUI in this figure creates three radio buttons, labeled "Option 1," "Option 2," and "Option 3," plus a text field to display the currently selected results.

The corresponding callback functions are shown in Figure 10.20*b*. When the user clicks on a radio button, the corresponding callback function is executed. That function sets the text box to display the current option, turns on that radio button, and turns off all other radio buttons.

Figure 10.21 on page 468 shows this GUI after Option 2 has been selected.

Popup Menus

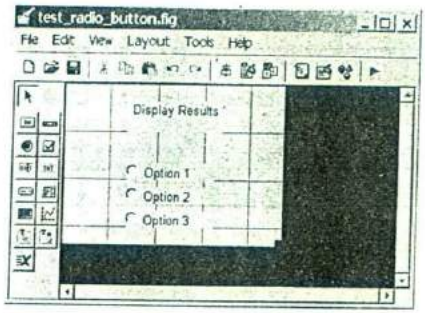
Popup menus are graphical objects that allow a user to select one of a mutually exclusive list of options. The list of options that the user can select among is specified by a cell array of strings, and the `'Value'` property indicates which of the strings is currently selected. A popup menu may be added to a GUI by using the popup menu tool ( ) in the Layout Editor.

Figure 10.22*a* on page 468 shows an example of a popup menu. This GUI in this figure creates a popup menu with five options, labeled "Option 1," "Option 2," and so forth.

The corresponding callback function is shown in Figure 10.22*b* on page 468. The callback function recovers the selected option by checking the `'Value'` parameter of the popup menu and creates and displays a string containing that value in the text field. Figure 10.23 on page 468 shows this GUI after Option 4 has been selected.

List Boxes

List boxes are graphical objects that display many lines of text and allow a user to select one or more of those lines. If there are more lines of text than can fit in



(a)

```
function Option1_Callback(hObject, eventdata, handles)

% Display the radio button clicked in the text field
set (handles.TextBox,'String','Option 1');

% Update all text fields
set (handles.Option1,'Value',1);
set (handles.Option2,'Value',0);
set (handles.Option3,'Value',0);

function Option2_Callback(hObject, eventdata, handles)

% Display the radio button clicked in the text field
set (handles.TextBox,'String','Option 2');

% Update all text fields
set (handles.Option1,'Value',0);
set (handles.Option2,'Value',1);
set (handles.Option3,'Value',0);

function Option3_Callback(hObject, eventdata, handles)

% Display the radio button clicked in the text field
set (handles.TextBox,'String','Option 3');

% Update all text fields
set (handles.Option1,'Value',0);
set (handles.Option2,'Value',0);
set (handles.Option3,'Value',1);
```

(b)

Figure 10.20 *(a)* Layout of a simple GUI with three radio buttons and a text field. *(b)* The callback functions for this GUI. When a user clicks on a radio button, it is set to 'on' and all other radio buttons are set to 'off'.

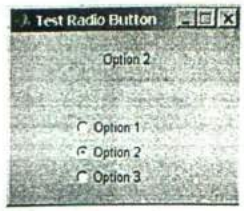
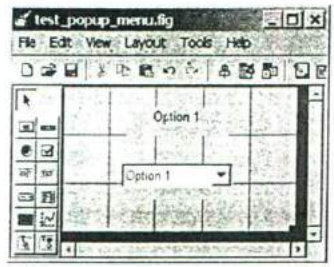


Figure 10.21 The GUI produced by program `test_radio_button` when Option 2 has been selected.



(a)

```
function Popup1_Callback(hObject, eventdata, handles)

% Find the value of the popup menu
value = get(handles.Popup1,'Value');

% Place the value into the text field
str = ['Option ' num2str(value)];
set (handles.Label1,'String',str);
```

(b)

Figure 10.22 *(a)* Layout of a simple GUI with a popup menu and a text field to display the current selection. *(b)* The callback functions for this GUI.

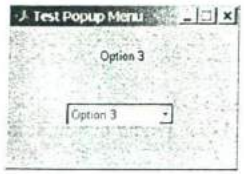


Figure 10.23 The GUI produced by program `test_popup_menu`.

the list box, a scroll bar will be created to allow the user to scroll up and down within the list box. The lines of text that the user can select among are specified by a cell array of strings, and the `'Value'` property indicates which of the strings are currently selected.

A list box is created by a `uicontrol` whose style property is `'listbox'`. A list box may be added to a GUI by using the listbox tool () in the Layout Editor.

List boxes can be used to select a single item from a selection of possible choices. In normal GUI usage, a single mouse click on a list item selects that item but does not cause an action to occur. Instead, the action waits on some external trigger, such as a pushbutton. However, a double mouse click causes an action to happen immediately. Single-click and double-click events can be distinguished using the `SelectionType` property of the figure in which the clicks occurred. A single mouse click will place the string `'normal'` in the `SelectionType` property, and a double mouse click will place the string `'open'` in the `SelectionType` property.

It is also possible for a list box to allow multiple selections from the list. If the difference between the `max` and `min` properties of the list box is greater than one, multiple selection is allowed. Otherwise, only one item may be selected from the list.

Figure 10.24*a* on page 470 shows an example of a single-selection list box. The GUI in this figure creates a list box with eight options, labeled "Option 1," "Option 2," and so forth. In addition, it creates a pushbutton to perform selection and a text field to display the selected choice. Both the list box and the pushbutton generate callbacks.

The corresponding callback functions are shown in Figure 10.24*b* on page 470. If a selection is made in the list box, function `Listbox1_Callback` will be executed. This function will check the *figure producing the callback* (using function `gcbf`) to see whether the selecting action was a single-click or a double-click. If it was a single-click, the function does nothing. If it was a double-click, then the function gets the selected value from the listbox and writes an appropriate string into the text field.

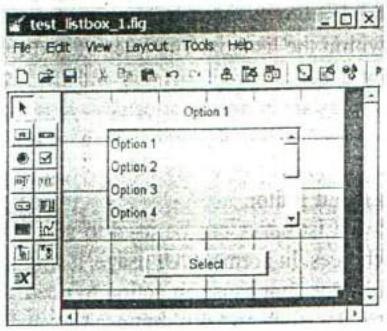
If the pushbutton is selected, function `Button1_Callback` will be executed. This function gets the selected value from the listbox and writes an appropriate string into the text field.

In an end-of-chapter exercise, you will be asked to modify this example to allow multiple selections in the list box.

Sliders

Sliders are graphical objects that allow a user to select values from a continuous range between a specified minimum value and a specified maximum value by moving a bar with a mouse. The `'Value'` property of the slider is set to a value between `min` and `max` depending on the position of the slider.

A slider is created by a `uicontrol` whose style property is `'slider'`. A slider may be added to a GUI by using the slider tool () in the Layout Editor.



(a)

```
function Button1_Callback(hObject, eventdata, handles)

% Find the value of the popup menu
value = get(handles.Listbox1,'Value');

% Update text label
str = ['Option ' num2str(value)];
set (handles.Label1,'String',str);


function Listbox1_Callback(hObject, eventdata, handles)

% If this was a double click, update the label.
selectiontype = get(gcbf,'SelectionType');
if selectiontype(1) == 'o'

   % Find the value of the popup menu
   value = get(handles.Listbox1,'Value');

   % Update text label
   str = ['Option ' num2str(value)];
   set (handles.Label1,'String',str);
end
```

(b)

Figure 10.24 *(a)* Layout of a simple GUI with a list box, a pushbutton, and a text field. *(b)* The callback functions for this GUI.

Figure 10.26*a* shows the layout for a simple GUI containing a slider and a text field. The `'Min'` property for this slider is set to zero, and the `'Max'` property is set to one. When a user drags the slider, it automatically calls the function `Slider1_Callback`, which is shown in Figure 10.26*b*. This function get the value of the slider from the `'Value'` property and displays the value in the text field. Figure 10.27 shows this GUI with the slider at some intermediate position in its range.

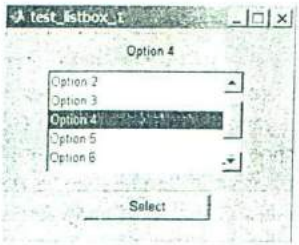
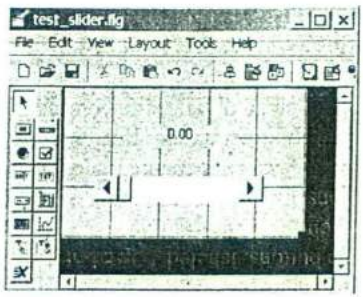


Figure 10.25 The GUI produced by program `test_listbox`.



(a)

```
function Slider1_Callback(hObject, eventdata, handles)

% Find the value of the slider
value = get(handles.Slider1,'Value');

% Place the value in the text field
str = sprintf('%.2f',value);
set (handles.Label1,'String',str);
```

(b)

Figure 10.26 *(a)* Layout of a simple GUI with a slider and a text field. *(b)* The callback function for this GUI.



Figure 10.27 The GUI produced by program `test_slider`.

Example 10.1—Temperature Conversion

Write a program that converts temperature from degrees Fahrenheit to degrees Celsius and vice versa over the range 0–100°C, using a GUI to accept data and display results. The program should include an edit box for the temperature in degrees Fahrenheit, an edit box for the temperature in degrees Celsius, and a slider to allow for the continuous adjustment of temperature. The user should be able to enter temperatures in either edit box or by moving the slider, and all GUI elements should adjust to the corresponding values.

SOLUTION To create this program, we will need a text field and an edit box for the temperature in degrees Fahrenheit, another text field and an edit box for the temperature in degrees Celsius, and a slider. We will also need a function to convert degrees Fahrenheit to degrees Celsius and a function to convert degrees Celsius to degrees Fahrenheit. Finally, we will need to write a callback function to support user inputs.

The range of values to convert will be 32–212°F or 0–100°C, so it will be convenient to set up the slider to cover the range 0–100, and to treat the value of the slider as a temperature in degrees C.

The first step in this process is to use `guide` to design the GUI. We can use `guide` to create the five required GUI elements and locate them in approximately the correct positions. Then, we can use the Property Inspector to perform the following steps:

1. Select appropriate names for each GUI element and store them in the appropriate `Tag` properties. The names will be `'Label1'`, `'Label2'`, `'Edit1'`, `'Edit2'`, and `'Slider1'`.
2. Store `'Degrees F'` and `'Degrees C'` in the `String` properties of the two labels.
3. Set the slider's minimum and maximum limits to 0 and 100, respectively.
4. Store initial values in the `String` property of the two edit fields and in the `Value` property of the slider. We will initialize the temperature to 32°F or 0°C, which corresponds to a slider value of 0.
5. Set the `Name` property of the figure containing the GUI to `'Temperature Conversion'`.

Once these changes have been made, the GUI should be saved to file `temp_conversion.fig`. This will produce both both a figure file and a matching M-file. The M-file will contain stubs for the three callback functions needed by the edit fields and the slider. The resulting GUI is shown during the layout process in Figure 10.28.

The next step in the process is to create the functions to convert degrees Fahrenheit to degrees Celsius. Function `to_c` will convert temperature from degrees Fahrenheit to degrees Celsius. It must implement the equation

$$\deg C = \frac{5}{9}(\deg F - 32) \tag{10-1}$$

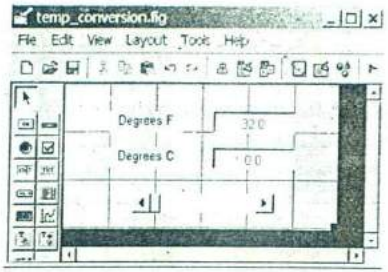


Figure 10.28 Layout of the temperature conversion GUI.

The code for this function is

```
function deg_c = to_c(deg_f)

% Convert degrees Fahrenheit to degrees C.
deg_c = (5/9) * (deg_f - 32);
end % function deg_c
```

Function `to_f` will convert temperature from degrees Celsius to degrees Fahrenheit. It must implement the equation

$$\deg F = \frac{9}{5} \deg C + 32 \tag{10-2}$$

The code for this function is

```
function deg_f = to_f(deg_c)

% Convert degrees Celsius to degrees Fahrenheit.
deg_f = (9/5) * deg_c + 32;
end % function deg_f
```

Finally, we must write the callback functions to tie it all together. The functions must respond to either the edit box or to the slider and must update all three components. (Note that we will update even the edit box that the user types into, so that the data can be displayed with a consistent format at all times as well as to correct errors if the user types an out-of-range input value.)

There is an extra complication here, because the values entered into edit boxes are *strings* and we wish to treat them as *numbers*. If a user types the value 100 into an edit box, he or she has really created the string `'100'`, not the number 100. The callback function must convert the strings into numbers so that the conversion can be calculated. This conversion is done with the `str2num` function, which converts a string into a numerical value.

Also, the callback function will have to limit user entries to the valid temperature range, which is 0–100°C and 32–212°F.

The resulting callback functions are shown in Figure 10.29 on page 474.

```
function Edit1_Callback(hObject, eventdata, handles)

% Update all temperature values
deg_f = str2num( get(hObject,'String') );
deg_f = max( [ 32 deg_f] );
deg_f = min( [212 deg_f] );
deg_c = to_c(deg_f);

% Now update the fields
set (handles.Edit1,'String',sprintf('%.1f',deg_f));
set (handles.Edit2,'String',sprintf('%.1f',deg_c));
set (handles.Slider1,'Value',deg_c);


function Edit2_Callback(hObject, eventdata, handles)

% Update all temperature values
deg_c = str2num( get(hObject,'String') );
deg_c = max( [  0 deg_c] );
deg_c = min( [100 deg_c] );
deg_f = to_f(deg_c);

% Now update the fields
set (handles.Edit1,'String',sprintf('%.1f',deg_f));
set (handles.Edit2,'String',sprintf('%.1f',deg_c));
set (handles.Slider1,'Value',deg_c);


function Slider1_Callback(hObject, eventdata, handles)

% Update all temperature values
deg_c = get(hObject,'Value');
deg_f = to_f(deg_c);

% Now update the fields
set (handles.Edit1,'String',sprintf('%.1f',deg_f));
set (handles.Edit2,'String',sprintf('%.1f',deg_c));
set (handles.Slider1,'Value',deg_c);
```

Figure 10.29 Callback functions for the temperature conversion GUI.

The program is now complete. Execute it and enter several different values using both the edit boxes and the sliders. Be sure to use some out-of-range values. Does it appear to be functioning properly?

◀

10.5 Additional Containers: Panels and Button Groups

MATLAB GUIs include two other types of containers: **panels** (created by the function `uipanel`) and **button groups** (created by the function `uibuttongroup`).

Panels

Panels are containers that can contain components or other containers, but they do *not* have a title bar and cannot have menus attached. A panel can contain GUI elements such as uicontrols, axes, other panels, or button groups. Any elements placed in a panel will be positioned relative to the panel. If the panel is moved on the GUI, then all of the elements within it are moved as well. Panels provide a great way to group related controls on a GUI.

A panel is created by a `uipanel` function. It can be added to a GUI by using the panel tool ( ) in the Layout Editor.

Each panel has a title, and is usually surrounded by an etched or beveled line marking the edges of the panel. The title of a panel can be located at the left, center, or right side of either the top or bottom of the panel. Samples of panels with several combinations of title positions and edge styles are shown in Figure 10.30.

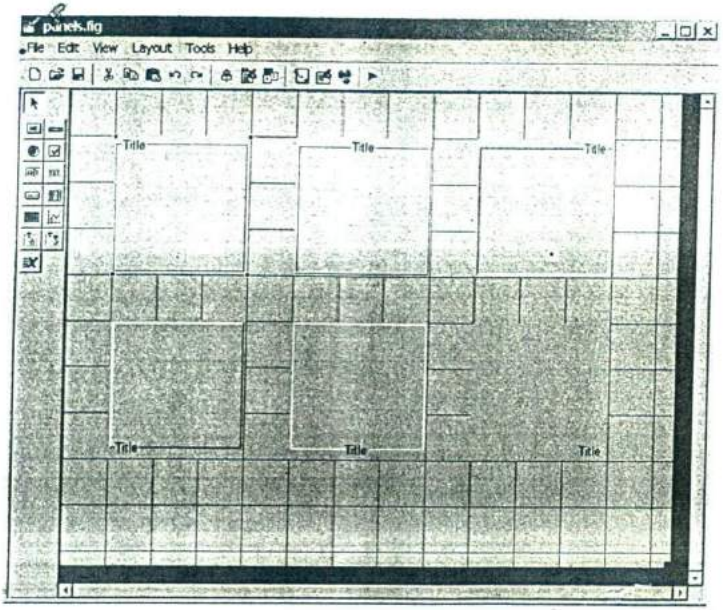


Figure 10.30 Examples of various panel styles.

Let's look at a simple example using panels. Suppose that we wanted to create a GUI to plot the function $y = ax^2 + bx + c$ between two specified values x_{min} and x_{max}. The GUI should allow the user to specify the values a, b, c, x_{min}, and x_{max}. In addition, it should allow the user to specify the style, color, and thickness of the line being plotted. These two sets of values (the ones specifying the line and the ones specifying what the line looks like) are logically distinct, so we can group them together in two panels on the GUI. One possible layout is shown in Figure 10.31. (You will be asked to finish this GUI and create an operational program in Exercise 10.7 at the end of the chapter.)

Table 10.4 contains a list of some important `uipanel` properties. These properties can be modified by the Property Inspector during the design phase, or they can be modified during execution with `get` and `set` functions.

Button Groups

Button groups are a special type of panel that can manage groups of radio buttons or toggle buttons to ensure that *no more than one button in the group is on at any time*. A button group is just like any other panel, except that the button group ensures that at most one radio button or toggle button is on at any given time. If one of them is turned on, the button group turns off any buttons that were already on.

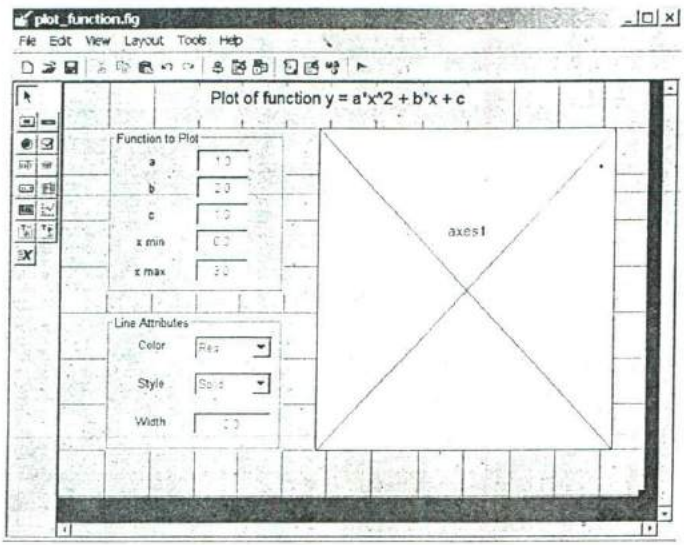


Figure 10.31 Layout of the Plot Function GUI, using panels to group related characteristics together.

Table 10.4 Important `uipanel` and `uibuttongroup` Properties

Property	Description
`BackgroundColor`	Specifies the color of the `uipanel` background. The value is either a predefined color such as `'r'`, `'g'`, or `'b'`, or else a 3-element vector specifying the red, green, and blue components of the color on a 0–1 scale. For example, the color magenta would be specified by `[1 0 1]`.
`BorderType`	Type of border around the `uipanel`. Options are `'none'`, `'etchedin'`, `'etchedout'`, `'beveledin'`, `'beveledout'`, or `'line'`. The default border type is `'etchedin'`.
`BorderWidth`	Width of border around the `uipanel`.
`FontAngle`	A string containing the font angle for the title text. Possible values are `'normal'`, `'italic'`, and `'oblique'`.
`FontName`	A string containing the font name for the title text.
`FontSize`	A number specifying the font size for the title text.
`FontUnits`	The units in which the font size is defined. Possible choices are `'inches'`, `'centimeters'`, `'normalized'`, `'points'`, and `'pixels'`. The default font units are `'points'`.
`FontWeight`	A string containing the font weight for the title text. Possible values are `'light'`, `'normal'`, `'demi'`, and `'bold'`. The default font weight is `'normal'`.
`ForegroundColor`	Specifies the color of the title font and the border.
`HighlightColor`	Specifies the 3D border highlight color.
`Position`	Specifies the position of a panel relative to its parent `figure`, `uipanel`, or `uibuttongroup`, in the units specified by the `'units'` property. This value accepts a 4-element vector in which the first two elements are the x and y positions of the lower left-hand corner of the panel, and the next two elements are the width and height of the panel.
`ShadowColor`	Specifies the color of the 3D border shadow.
`Tag`	The "name" of the `uipanel`, which can be used to access it.
`Title`	The title string.
`TitlePosition`	Location of the title string on the `uipanel`. Possible values are `'lefttop'`, `'centertop'`, `'righttop'`, `'leftbottom'`, `'centerbottom'`, and `'rightbottom'`. The default value is `'lefttop'`.
`Units`	The units used to describe the position of the `uipanel`. Possible choices are `'inches'`, `'centimeters'`, `'normalized'`, `'points'`, `'pixels'`, or `'characters'`. The default units are `'normalized'`.
`Visible`	Specifies whether or not this `uipanel` is visible. Possible values are `'on'` or `'off'`.

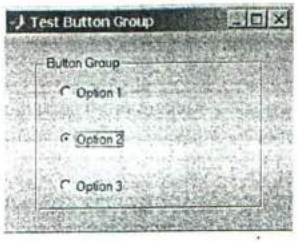


Figure 10.32 A button group controlling three radio buttons.

A button group is created by a `uibuttongroup` function. It can be added to a GUI by using the button group tool ( ) in the Layout Editor.

If a radio button or a toggle button is controlled by a Button Group, the user must attach the name of the function to execute when that button is selected in a special property called `SelectionChangeFcn`. This callback is executed by the GUI whenever a radio button or toggle button changes state. Do *not* place the function in the usual `Callback` property, because the Button Group overwrites the callback property for every radio button or toggle button that it controls.

Figure 10.32 shows shows a simple GUI containing a Button Group and three radio buttons, labeled `'Option 1'`, `'Option 2'`, and `'Option 3'`. When a user clicks on one radio button in the group, the button is turned on and all other buttons in the group are turned off.

10.6 Dialog Boxes

A **dialog box** is a special type of figure that is used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. They are also used to select files or printer properties.

Dialog boxes may be **modal** or **non-modal**. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed, while a normal dialog box does not block access to other windows. Modal dialog boxes are typically used for warning and error messages that need urgent attention and cannot be ignored. By default, most dialog boxes are non-modal.

MATLAB includes many types of dialog boxes, the more important of which are summarized in Table 10.5. We will examine only a few types of dialog boxes here, but you can consult the MATLAB on-line documentation for the details of the others.

Table 10.5 Selected Dialog Boxes

Property	Description
`dialog`	Creates a generic dialog box.
`errordlg`	Displays an error message in a dialog box. The user must click the `OK` button to continue.
`helpdlg`	Displays a help message in a dialog box. The user must click the `OK` button to continue.
`inputdlg`	Displays a request for input data and accepts the user's input values.
`listdlg`	Allows a user to make one or more selections from a list.
`printdlg`	Displays a printer-selection dialog box.
`questdlg`	Asks a question. This dialog box can contain either two or three buttons, which by default are labeled `Yes`, `No`, and `Cancel`.
`uigetfile`	Displays a file-open dialog box. This box allows a user to select a file to open, *but does not actually open the file.*
`uiputfile`	Displays a file-save dialog box. This box allows a user to select a file to save, *but does not actually save the file.*
`uisetcolor`	Displays a color-selection dialog box.
`uisetfont`	Displays a font-selection dialog box.
`warndlg`	Displays a warning message in a dialog box. The user must click the `OK` button to continue.

Error and Warning Dialog Boxes

Error and warning dialog boxes have similar calling parameters and behavior. In fact, the only difference between them is the icon displayed in the dialog box. The most common calling sequence for these dialog boxes is

```
errordlg(error_string,box_title,create_mode);
warndlg(warning_string,box_title,create_mode);
```

The `error_string` or `warning_string` is the message to display to the user, and the `box_title` is the title of the dialog box. Finally, `create_mode` is a string that can be `'modal'` or `'non-modal'`, depending on the type of dialog box you wish to create.

For example, the following statement creates a modal error message that cannot be ignored by the user. The dialog box produced by this statement is shown in Figure 10.33.

```
errordlg('Invalid input values!','Error Dialog Box','modal');
```

Input Dialog Boxes

Input dialog boxes prompt a user to enter one or more values that may be used by a program. They may be created with one of the following calling sequences.

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,title)
answer = inputdlg(prompt,title,line_no)
answer = inputdlg(prompt,title,line_no,default_answer)
```

Here, `prompt` is a cell array of strings, with each element of the array corresponding to one value that the user will be asked to enter. The parameter `title` specifies the title of the dialog box, while `line_no` specifies the number of lines to be allowed for each answer. Finally, `default_answer` is a cell array containing the default answers that will be used if the user fails to enter data for a particular item. Note that there must be as many default answers as there are prompts.

When the user clicks the OK button on the dialog box, his or her answers will be returned as a cell array of strings in variable `answer`.

As an example of an input dialog box, suppose that we wanted to allow a user to specify the position of a figure using an input dialog. The code to perform this function would be

```
prompt{1} = 'Starting x position:';
prompt{2} = 'Starting y position:';
prompt{3} = 'Width:';
prompt{4} = 'Height:';
title = 'Set Figure Position';
default_ans = {'50','50','180','100'};
answer = inputdlg(prompt,title,1,default_ans);
```

The resulting dialog box is shown in Figure 10.34.

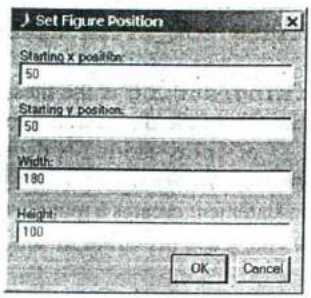


Figure 10.34 An input dialog box.

The uigetfile, uisetfile, and uigetdir Dialog Boxes

The uigetfile and uisetfile dialog boxes allow a user to interactively pick files to open or save. These functions use the standard file-open or file-save dialog boxes for the particular operating system that MATLAB is running on. They return strings containing the name and the path of the file but do not actually read or save it. The programmer is responsible for writing additional code for that purpose.

The form of these two dialog boxes is

```
[filename, pathname] = uigetfile(filter_spec,title);
[filename, pathname] = uisetfile(filter_spec,title);
```

Parameter filter_spec is a string specifying the type of files to display in the dialog box, such as '*.m', '*.mat', etc. Parameter title is a string specifying the title of the dialog box. After the dialog box executes, filename contains the name of the selected file and pathname contains the path of the file. If the user cancels the dialog box, filename and pathname are set to zero.

The following script file illustrates the use of these dialog boxes. It prompts the user to enter the name of a Mat-file and then reads the contents of that file. The dialog box created by this code on a Windows 2000 Professional system is shown in Figure 10.35. (This is the standard open-file dialog for Windows 2000 Professional. It will appear slightly different on Windows NT 4.0, Windows XP, or Unix Linux systems).

```
[filename, pathname] = uigetfile('*.mat','Load MAT File');
if filename ~= 0
   load([pathname filename]);
end
```

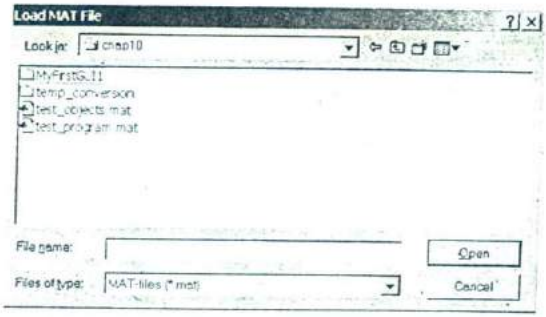


Figure 10.35 A file open dialog box created by uigetfile.

The `uigetdir` dialog box allows a user to interactively select a directory. This function uses the standard directory-selection dialog box for the particular operating system that MATLAB is running on. It returns the name of the directory but does not actually do anything with it. The programmer is responsible for writing additional code to use the directory name.

The form of this dialog box is

```
directoryname = uigetdir(start_path, title);
```

Parameter `start_path` is the path of the initially selected directory. If it is not valid, the dialog box opens with the base directory selected. Parameter `title` is a string specifying the title of the dialog box. After the dialog box executes, `directoryname` contains the name of the selected directory. If the user cancels the dialog box, `directoryname` is set to zero.

The following script file illustrates the use of this dialog box. It prompts the user to select a directory starting with the current MATLAB working directory. This dialog box created by this code on a Windows 2000 Professional system is shown in Figure 10.36. (The style of the box varies for different operating systems. It will appear slightly different in Windows NT 4.0, Windowx XP, or Unix/Linux systems).

```
dir1 = uigetdir('d:\matlab7\work','Select a directory');
if dir1 ~= 0
   cd(dir1);
end
```

The `uisetcolor` and `uisetfont` Dialog Boxes

The `uisetcolor` and `uisetfont` dialog boxes allow a user to interactively select colors or fonts using the standard dialog boxes for the computer on which



Figure 10.36 A directory selection dialog box created by `uigetdir`.

MATLAB is executing. The appearances of these boxes will vary for different operating systems. They provide a standard way to select colors or fonts within a MATLAB GUI.

Consult the MATLAB on-line documentation to learn more about these special-purpose dialog boxes. We will use them in some of the end-of-chapter exercises.

Good Programming Practice

Use dialog boxes to provide information or request input in GUI-based programs. If the information is urgent and should not be ignored, make the dialog boxes modal.

10.7 Menus

Menus can also be added to MATLAB GUIs. A menu allows a user to select actions without additional components appearing on the GUI display. They are useful for selecting less commonly used options without cluttering up the GUI with a lot of extra buttons.

There are two types of menus in MATLAB: **standard menus**, which are pulled down from the menu bar at the top of a figure, and **context menus**, which pop up over the figure when a user right-clicks the mouse over a graphical object. We will learn how to create and use both types of menus in this section.

Standard menus are created with `uimenu` objects. Each item in a menu is a separate `uimenu` object, including items in submenus. These `uimenu` objects are similar to `uicontrol` objects, and they have many of the same properties such as `Parent`, `Callback`, `Enable`, and so forth. A list of the more important `uimenu` properties is given in Table 10.6.

Each menu item is attached to a parent object, which is a figure for the top-level menus, or another menu item for submenus. All of the `uimenus` connected to the same parent appear on the same menu, and the cascade of items forms a tree of submenus. Figure 10.37*a* (on page 485) shows a typical MATLAB menu in operation, and Figure 10.37*b* (on page 486) shows the relationship among the objects making up the menu.

MATLAB menus are created using the Menu Editor, which can be selected by clicking the ( ) icon on the toolbar in the `guide` Layout Editor. Figure 10.37*c* shows the Menu Editor with the menu items that generate this menu structure. The additional properties in Table 10.6 that are not shown in the Menu Editor can be set with the Property Editor (`propedit`).

Top-level context menus are created by `uicontextmenu` objects, and the lower level items within context menus are created by `uimenu` objects. Context

Table 10.6 Important uimenu Properties

Property	Description
`Accelerator`	A single character specifying the keyboard equivalent for the menu item. The keyboard combination CTRL+key allows a user to activate the menu item from the keyboard.
`Callback`	Specifies the name and parameters of the function to be called when the menu item is activated. It the menu item has a submenu, the callback executes *before the submenu is displayed*. If the menu item does not have submenus, the callback executes when the mouse button is *released*.
`Checked`	When this property is `'on'`, a checkmark is placed to the left of the menu item. This property can be used to indicate the status of menu items that toggle between two states. Possible values are `'on'` or `'off'`.
`Enable`	Specifies whether or not this menu item is selectable. If it not enabled, the menu item will not respond to mouse clicks or accelerator keys. Possible values are `'on'` or `'off'`.
`ForegroundColor`	Set color of text in the menu item.
`Label`	Specifies the text to be displayed on the menu. The ampersand character (&) can be used to specify a keyboard mnemonic for this menu item; it will not appear on the label. For example, the string `'&File'` will create a menu item displaying the text `'File'` and responding to the F key.
`Parent`	The handle of the parent object for this menu item. The parent object could be a figure or another menu item.
`Position`	Specifies the position of a menu item on the menu bar or within a menu. Position 1 is the left-most menu position for a top-level menu, and the highest position within a submenu.
`Separator`	When this property is `'on'`, a separating line is drawn above this menu item. Possible values are `'on'` or `'off'`.
`Tag`	The "name" of the menu item, which can be used to access it.
`Visible`	Specifies whether or not this menu item is visible. Possible values are `'on'` or `'off'`.

menus are basically the same as standard menus, except that they can be associated with any GUI object (e.g., axes, lines, text, figures).

Suppressing the Default Menu

Every MATLAB figure comes with a default set of standard menus. If you wish to delete these menus from a figure and create your own menus, you must first turn the default menus off. The display of default menus is controlled by the figure's `MenuBar` property. The possible values of this property are `'figure'`

Table 10.7 Important `uicontextmenu` Properties

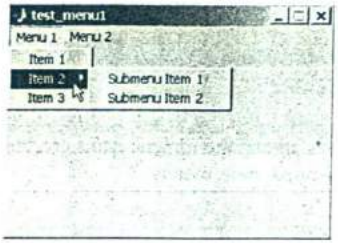
Property	Description
`Callback`	Specifies the name and parameters of the function to be called when the context menu is activated. The callback executes before the context menu is displayed.
`Parent`	The handle of the parent object for this context menu.
`Tag`	The "name" of the context menu, which can be used to access it.
`Visible`	Specifies whether or not this context menu is visible. This property is set automatically, and should normally not be modified.

and `'none'`. If the property is set to `'figure'`, the default menus are displayed. If the property is set to `'none'`, the default menus are suppressed. You can use the Property Inspector to set the `MenuBar` property for your GUIs when you create them.

Creating Your Own Menus

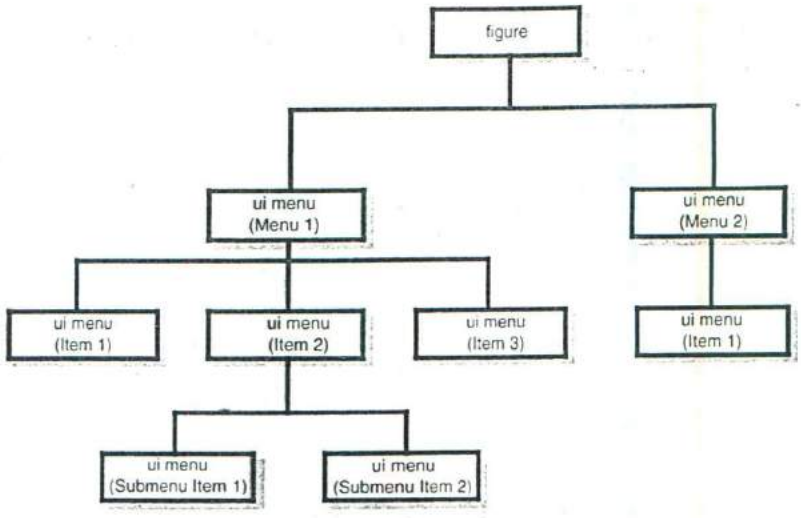
Creating your own standard menus for a GUI is basically a three-step process.

1. First, create a new menu structure with the Menu Editor. Use the Menu Editor to define the structure, giving each menu item a `Label` to display and a unique `Tag` value. You can also specify whether or not there is a separator bar between menu items, and whether or not each menu item has a check mark by it. A dummy callback function will be generated automatically for each menu item.

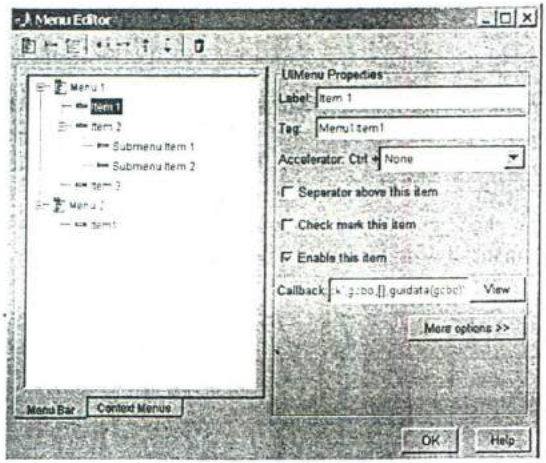


(a)

Figure 10.37 *(a)* A typical menu structure.



(b)



(c)

Figure 10.37 *(continued) (b)* The relationships among the uimenu items creating the menu. *(c)* The Menu Editor structure that generated these menus.

2. If necessary, edit the properties of each menu item using the Property Inspector. The Property Inspector can be started by clicking the "More Options" button on the Menu Editor. Because the most important menu item properties (Label, Tag, Callback, Checked, and Separator) can be set on the Menu Editor, the Property Inspector is usually not needed. However, if you must set any of the other properties listed in Table 10.6, you will need to use the Property Inspector.
3. Third, implement a callback function to perform the actions required by your menu items. The prototype function is created automatically, but you must add the code to make each menu item behave properly.

The process of building menus will be illustrated in an example at the end of this section.

Programming Pitfalls

Only the `Label`, `Tag`, `Callback`, `Checked`, and `Separator` properties of a menu item can be set from the Menu Editor. If you need to set any of the other properties, you will have to use the Property Inspector on the figure and select the appropriate menu item to edit.

Accelerator Keys and Keyboard Mnemonics

MATLAB menus support accelerator keys and keyboard mnemonics. **Accelerator keys** are "CTRL+key" combinations that cause a menu item to be executed *without opening the menu first*. For example, the accelerator key "o" might be assigned to the File/Open menu item. In that case, the keyboard combination CTRL+o will cause the File Open callback function to be executed.

A few CRTL+key combinations are reserved for the use of the host operating system. These combinations differ between PC and Unix systems; consult the MATLAB on-line documentation to determine which combinations are legal for your type of computer.

Accelerator keys are defined by setting the `Accelerator` property in a `uimenu` object.

Keyboard mnemonics are single letters that can be pressed to cause a menu item to execute once the menu is open. The keyboard mnemonic letter for a given menu item is underlined[1]. For top-level menus, the keyboard mnemonic is exe-

[1] On some versions of Windows, the underlines are hidden until the ALT key is held down. This behavior can be modified. For example, the underlines can be made visible all the time in Windows 2000 by going to the Display item in the Control Panel, selecting the Effects tab, and clearing the item "Hide keyboard navigation indicators until I use the Alt key"

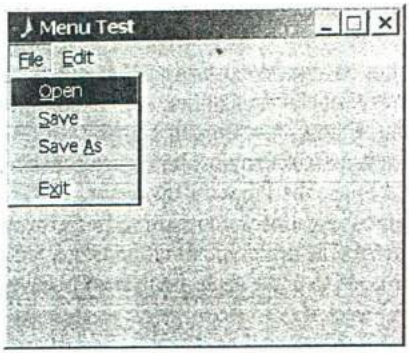


Figure 10.38 An example showing keyboard mnemonics. The menu shown was opened by typing the keys ALT+f, and the Exit option could be executed by simply typing 'x'.

cuted by pressing ALT plus the mnemonic key at the same time. Once the top level menu is open, simply pressing the mnemonic key will cause a menu item to execute.

Figure 10.38 illustrates the use of keyboard mnemonics. The File menu is opened with the keys ALT+f, and once it is opened the Exit menu item can be executed by simply typing "x."

Keyboard mnemonics are defined by placing the ampersand character (&) before the desired mnemonic letter in the `Label` property. The ampersand will not be displayed, but the following letter will be underlined and it will act as a mnemonic key. For example, the `Label` property of the Exit menu item in Figure 10.29 is `'E&xit'`.

Creating Context Menus

Context menus are created in the same fashion as ordinary menus, except that the top-level menu item is a `uicontextmenu`. The parent of a `uicontextmenu` must be a figure, but the context menu can be associated with and respond to right mouse clicks on any graphical object. Context menus are created using the "Context Menu" selection on the Menu Editor. Once the context menu is created, any number of menu items can be created under it.

To associate a context menu with a specific object, you must set the object's `UIContextMenu` property to the handle of the `uicontextmenu`. This is normally done using the Property Inspector, but it can be done with the `set` command as shown below. If `Hcm` is the handle to a context menu, the following statements will associate the context menu with a line created by a `plot` command.

```
H1 = plot(x,y);
set (H1,'UIContextMenu',Hcm);
```

We will create a context menu and associate it with a graphical object in the following example.

Example 10.2—Plotting Data Points

Write a program that opens a user-specified data file and plots the line specified by the points in the file. The program should include a File menu, with Open and Exit menu items. The program should also include a context menu attached to the line, with options to change the line style. Assume that the data in the file is in the form of (x, y) pairs, with one pair of data values per line.

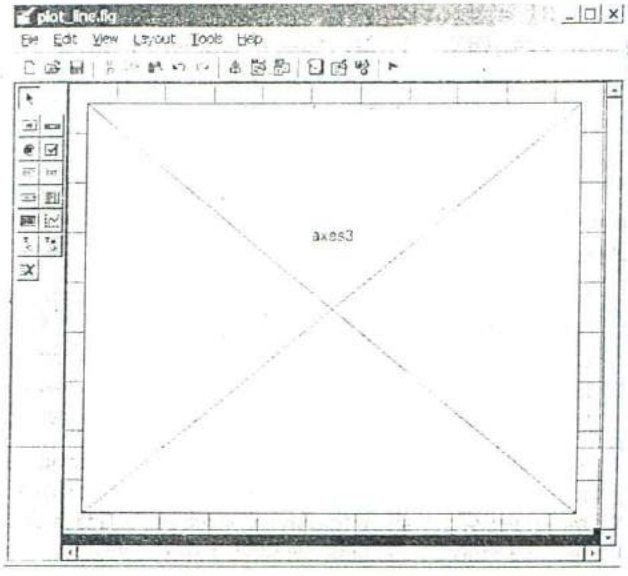
SOLUTION This program should include a standard menu with Open and Exit menu items plus a set of axes on which to plot the data. It should also include a context menu specifying various line styles, which can be attached to the line after it is plotted. The options should include solid, dashed, dotted, and dash-dot line styles.

The first step in creating this program is to use `guide` to create the required GUI, which is only a set of axes in this case (see Figure 10.39*a* on page 490). Then, we must use the Menu Editor to create the File menu. This menu will contain Open and Exit menu items, as shown in Figure 10.39*b* on page 490. Note that we must use the Menu Editor to set the `Label` and `Tag` and strings for each of these menu items. We will also define keyboard mnemonics "F" for File, "O" for Open, and "x" for Exit, and place a separator between the Open and Exit menu items. Figure 10.39*b* shows the Exit menu item. Note that "x" is the keyboard mnemonic, and that the separator switch is turned on.

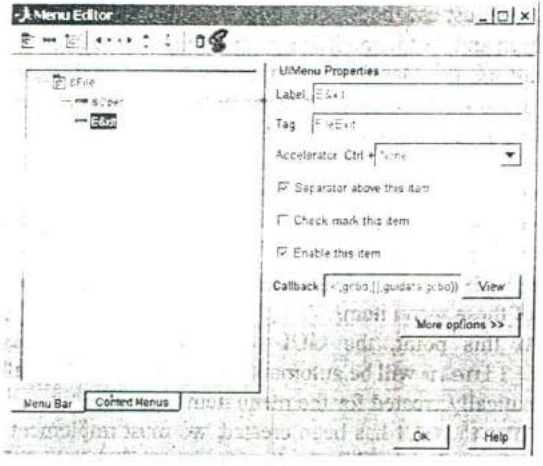
Next, we must use the Menu Editor to create the context menu. This menu starts with a `uicontextmenu` object, with four menu items attached to it (see Figure 10.39*c* on page 491). Again, we must set the `Label` and `Tag` strings for each of these menu items.

At this point, the GUI should be saved as `plot_line.fig`, and `plot_line.m` will be automatically created. Dummy callback functions will be automatically created for the menu items.

After the GUI has been created, we must implement six callback functions for the Open, Exit, and linestyle menu items. The most difficult callback function is the response to the File/Open menu item. This callback must prompt the user for the name of the file (using a `uigetfile` dialog box), open the file, read the data, save it into `x` and `y` arrays, and close the file. Then, it must plot the line and save the line's handle as application data so that we can use it to modify the line style later. Finally, it must associate the context menu with the line. The `FileOpen_Callback` function is shown in Figure 10.40 on page 491. Note that the function uses a dialog box to inform the user of file-open errors.

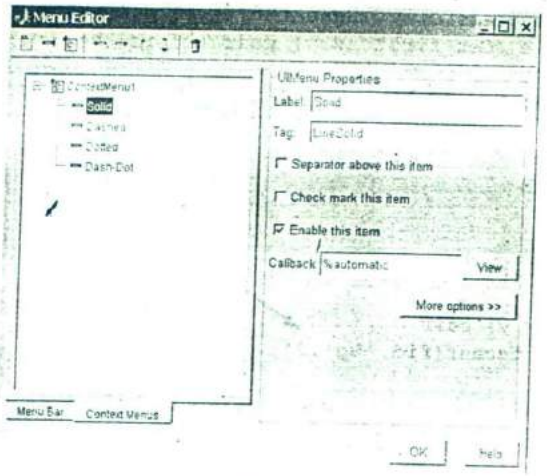


(a)



(b)

Figure 10.39 *(a)* The layout for `plot_line`. *(b)* The File menu in the Menu Editor.



(c)

Figure 10.39 *(continued)* (c) The context menu in the Menu Editor.

```
function varargout = FileOpen_Callback(h, eventdata, ...
                     handles, varargin)

% Get the file to open
[filename, pathname] = uigetfile('*.dat','Load Data');
if filename ~= 0

   % Open the input file
   filename = [pathname filename];
   [fid,msg] = fopen(filename,'rt');

   % Check to see of the open failed.
   if fid < 0

      % There was an error--tell user.
      str = ['File ' filename ' could not be opened.'];
      title = 'File Open Failed';
      errordlg(str,title,'modal');

   else
```

Get file name to open

Open file

Error message if open fails

Figure 10.40 The File/Open callback function.

```
% File opened successfully. Read the (x,y) pairs from
% the input file. Get first (x,y) pair before the
% loop starts.
[in,count] = fscanf(fid,'%g',2);
ii = 0;

while ~feof(fid)
  ii = ii + 1;
   x(ii) = in(1);
   y(ii) = in(2);

  % Get next (x,y) pair
  [in,count] = fscanf(fid,'%g',2);
end

% Data read in. Close file.
fclose(fid);

% Now plot the data.
 hline = plot(x,y,'LineWidth',3);
 xlabel('x');
 ylabel('y');
grid on;

% Associate the context menu with line
set(hline,'Uicontextmenu',handles.ContextMenu1);

% Save the line's handle as application data
handles.hline = hline;
guidata(gcbf, handles);

end
```

Read data

Plot line

Set context menu

Save handle as app data

Figure 10.40 *(continued)*

The remaining callback functions are very simple. The `FileExit_Callback` function simply closes the figure, and the line style functions simply set the line style. When the user right-clicks a mouse button over the line, the context menu will appear. If the user selects an item from the menu, the resulting callback will use the line's saved handle to change its properties. These five functions are shown in Figure 10.41.

The final program is shown in Figure 10.42. Experiment with it on your own computer to verify that it behaves properly.

```
function varargout = FileExit_Callback(h, eventdata, ...
                                handles, varargin)
close(gcbf);


function varargout = LineSolid_Callback(h, eventdata, ...
                                handles, varargin)
set(handles.hline,'LineStyle','-');function varargout =
LineDashed_Callback(h, eventdata, ...
                                handles, varargin)
set(handles.hline,'LineStyle','--');


function varargout = LineDotted_Callback(h, eventdata, ...
                                handles, varargin)
set(handles.hline,'LineStyle',':');


function varargout = LineDashDot_Callback(h, eventdata, ...
                                handles, varargin)
set(handles.hline,'LineStyle','-.');
```

Figure 10.41 The remaining callback functions in `plot_line`.

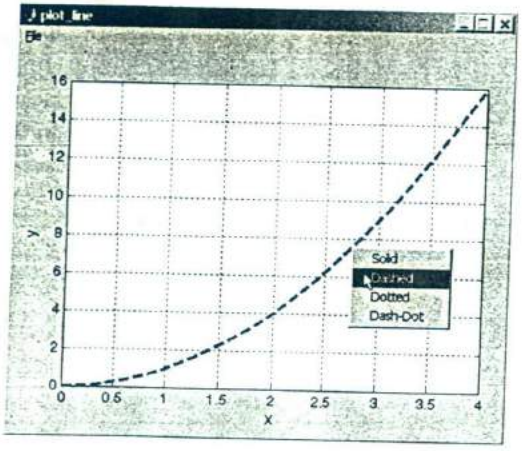


Figure 10.42 The GUI produced by program `plot_line`.

Quiz 10.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 10.1 through 10.7. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. List the types of graphical components discussed in this chapter. What is the purpose of each one?
2. List the types of containers discussed in this chapter. What are the differences among them?
3. What is a callback function? How are callback functions used in MATLAB GUIs?
4. Describe the steps required to create a GUI-based program.
5. Describe the purpose of the handles data structure.
6. How is application data saved in a MATLAB GUI? Why would you want to save application data in a GUI?
7. How can you make a graphical object invisible? How can you turn a graphical object off so that it will not respond to mouse clicks or keyboard input?
8. Which of the GUI components described in this chapter respond to mouse clicks? Which ones respond to keyboard inputs?
9. What are dialog boxes? How can you create a dialog box?
10. What is the difference between a modal and a non-modal dialog box?
11. What is the difference between a standard menu and a context menu? What components are used to create these menus?
12. What are accelerator keys? What are mnemonics?

10.8 Tips for Creating Efficient GUIs

This section lists a few miscellaneous tips for creating efficient graphical user interfaces.

Tool Tips

MATLAB GUIs support **tool tips**, which are small help windows that pop up beside a `uicontrol` GUI object whenever the mouse is held over the object for a while. Tool tips are used to provide a user with quick help about the purpose of each object on a GUI.

A tool tip is defined by setting an object's `TooltipString` property to the string that you wish to display. You will be asked to create tool tips in the end of chapter exercises.

Good Programming Practice

Define tool tips to provide users with helpful hints about the functions of your GUI components.

Pcode

The first time MATLAB executes a function during a program's execution, it compiles (or parses) the function into an intermediate code called **pcode** (which is short for pseudocode) and then executes the pcode in its run-time interpreter. Once a function has been compiled, it remains in MATLAB's memory, and it can be executed over and over again without having to be re-compiled. However, the next time MATLAB executes, the function will have to be compiled again.

The penalty associated with this initial compilation is relatively small, but it becomes more and more significant as function sizes get larger. Since the functions that define a GUI are typically quite large, the compilation penalty for GUI-based programs is relatively larger than for other types of programs. In other words, the GUIs run more slowly because of this initial compilation time.

Fortunately, there is a way to avoid this extra penalty. It is possible to precompile your MATLAB functions and script files into pcode and save the pcode in files for immediate execution in the future. Executing the pcode files saves the initial compilation time, and so makes your programs run faster.

MATLAB creates pcode files with the command `pcode`. This command takes one of the following forms:

```
pcode fun1.m fun2.m fun3.m ...
pcode *.m
```

The first form compiles the named files, and the second form compiles every M-file in the current directory. The compiled output is saved with the file extent "p." For example, if you compile the file `foo.m`, the compiled output will be stored in file `foo.p`.

If a function exists as both an M-file and a p-file, MATLAB will automatically execute the p-file version since it will be faster. However, if you modify the M-file, you *must* remember to manually recompile it, or the program will continue to execute your old code!

Compiling files to pcode can have an additional advantage as well. You can protect your investment in your source code by distributing the program to others

in the form of pcode files. They may be freely executed, but it is not easy for someone to re-engineer the files and take your ideas.

Good Programming Practice

Once a program is working properly, you may use the pcode command to precompile its M-files and increase the program's speed.

Programming Pitfalls

If you change an M-file that has been compiled into pcode, always remember to recompile the file. Otherwise, you will continue to execute your old, unmodified code!

Toolbars

MATLAB GUIs can also support *toolbars*. A toolbar is a row of special pushbuttons or toggle buttons along the top of a figure, just below the menu bar. Each button has a small figure or icon on it, representing its function. We have seen examples of toolbars in most of the MATLAB figures produced in this book. For example, Figure 10.43 shows a simple plot displaying the default toolbar.

Every figure has a `ToolBar` property, which determines whether or not the default figure toolbar is displayed. The possible values of this property are `'none'`, `'auto'`, and `'figure'`. If the property is `'none'`, the default

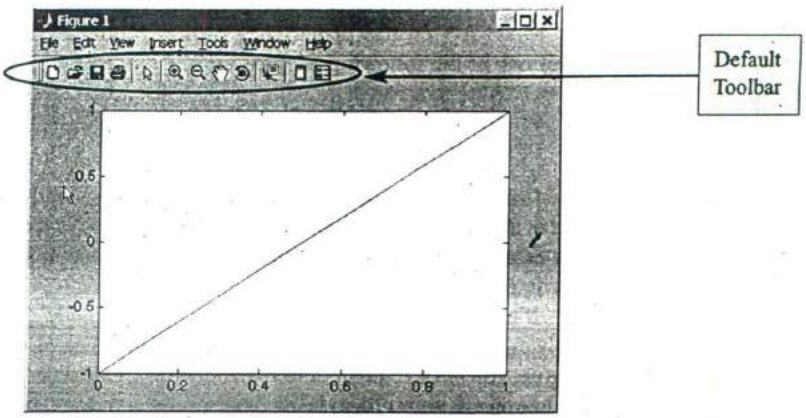


Figure 10.43 A MATLAB figure showing the default toolbar.

toolbar is not displayed. If the property is `'figure'`, the default toolbar is displayed. If the property is `'auto'`, the default toolbar is displayed unless the user defines a custom toolbar. If the property is `'auto'` and the user defines a custom toolbar, then it will be displayed instead of the default toolbar.

A programmer can create his or her own toolbar using the `uitoolbar` function, and can add the toolbar equivalent of pushbuttons and toggle buttons to the toolbar using the `uipushtool` and `uitoggletool` functions. The user-defined toolbar can be displayed in addition to or instead of the default figure toolbar.

Unfortunately, toolbars are not yet integrated into `guide`. If you want to use toolbars in your GUIs, you must create them manually. More details about these functions can be found in the MATLAB on-line help documentation.

Additional Enhancements

GUI-based programs can be much more sophisticated than we have described in this introductory chapter. In addition to the `Callback` property that we have been using in the chapter, `uicontrols` support four other types of callbacks: `CreateFcn`, `DeleteFcn`, `ButtonDownFcn`, and `KeyPressFcn`. MATLAB figures also support three important types of callbacks: `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`.

The **`CreateFcn`** property defines a callback that is automatically called whenever an object is created. It allows a programmer to customize his or her objects as they are created during program execution. Since this callback is executed before the object is completely defined, a programmer must specify the function to execute as a default property of the root before the object is created. For example, the following statement will cause the function `function_name` to be executed each time that a `uicontrol` is created. The function will be called after MATLAB creates the object's properties, so they will be available to the function when it executes.

```
set(0,'DefaultUicontrolCreateFcn','function_name')
```

The **`DeleteFcn`** property defines a callback that is automatically called whenever an object is destroyed. It is executed before the object's properties are destroyed, so they will be available to the function when it executes. This callback provides the programmer with an opportunity to do custom clean-up work.

The **`ButtonDownFcn`** property defines a callback that is automatically called whenever a mouse button is pressed within a 5-pixel border around a `uicontrol`. If the mouse button is pressed on the `uicontrol`, the `Callback` is executed. Otherwise, if it is near the border, the `ButtonDownFcn` is executed. If the `uicontrol` is not enabled, the `ButtonDownFcn` is executed even for clicks on the control.

The **`KeyPressFcn`** property defines a callback that is automatically called whenever a key is pressed while the specified object is selected or highlighted.

This function can find out which key was pressed by checking the `CurrentCharacter` property of the enclosing figure, or else by checking the contents of the event data structure passed to the callback. It can use this information to change behaviour, depending on which key was pressed.

The figure-level callback functions **`WindowButtonDownFcn`**, **`WindowButtonMotionFcn`**, and **`WindowButtonUpFcn`** allow a programmer to implement features such as animations and drag-and-drop, because the callbacks can detect the initial, intermediate, and final locations at which the mouse button is pressed. They are beyond the scope of this book, but are well worth learning about. Refer to the *Creating Graphical User Interfaces* section in the MATLAB user documentation for a description of these callbacks.

Example 10.3—Creating a Histogram GUI

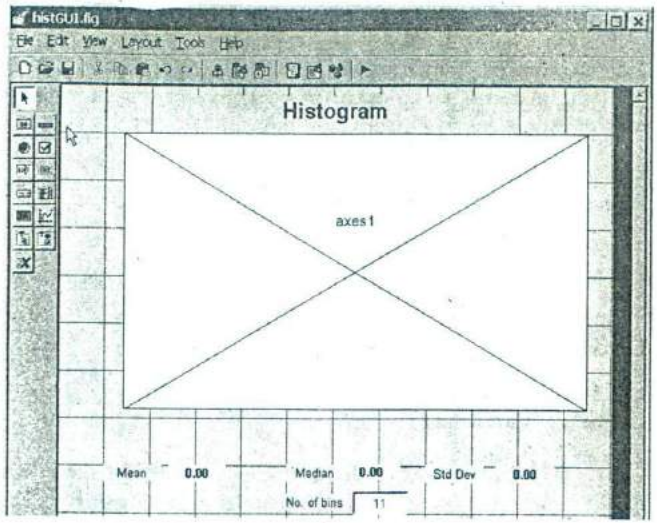
Write a program that opens a user-specified data file and calculates a histogram of the data in the file. The program should calculate the mean, median, and standard deviation of the data in the file. It should include a File menu, with Open and Exit menu items. It should also include a means to allow the user to change the number of bins in the histogram.

Select a color other than the default color for the figure and the text label backgrounds, use keyboard mnemonics for menu items, and add tool tips where appropriate.

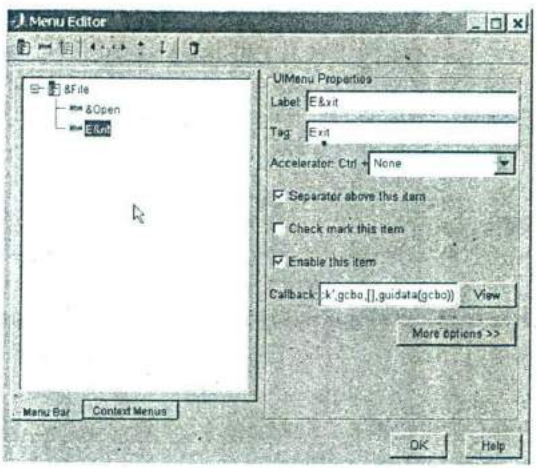
SOLUTION This program should include a standard menu with Open and Exit menu items, a set of axes on which to plot the histogram, and a set of six text fields for the mean, median, and standard deviation of the data. Three of these text fields will hold labels, and three will hold the read-only mean, median, and standard deviation values. It must also include a label and an edit field to allow the user to selected the number of bins to display in the histogram.

We will select a light blue color [0.6 1.0 1.0] for the background of this GUI. To make the GUI have a light blue background, this color vector must be loaded into the `'Color'` property property of the figure and into the `'BackgroundColor'` property of each text label with the Property Inspector during the GUI layout. Since MATLAB GUIs automatically reset their figure color to match the system default, we will also need to set this figure color in the `histGUI_OpeningFcn` callback, which is called just before the figure becomes visible.

The first step in creating this program is to use `guide` to lay out the required GUI (see Figure 10.44*a*). Then, use the Property Inspector to set the properties of the seven text fields and the edit field. The fields must be given unique tags so that we can locate them from the callback functions. Next, use the Menu Editor to create the File menu (see Figure 10.44*b*). Finally, the resulting GUI should be saved as `histGUI`, creating `histGUI.fig` and `histGUI.m`.



(a)



(b)

Figure 10.44 *(a)* The layout for `histGUI`. *(b)* The File menu in the Menu Editor.

After `histGUI.m` has been saved, the function `histGUI_OpeningFcn` must be edited to initialize the background color of the figure and to save the initial number of histogram bins in the `handles` structure. The modified code for the opening function is:

```
% --- Executes just before histGUI is made visible.
function histGUI_OpeningFcn(hObject, eventdata, handles, varargin)
```

```
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to histGUI (see VARARGIN)

% Choose default command line output for histGUI
handles.output = hObject;

% Set the figure background color
set(hObject,'Color',[0.6 1 1]);

% Set the initial number of bins
handles.nbins = 11;

% Update handles structure
guidata(hObject, handles);
```

Next, we must create callback functions for the File/Open menu item, the File/Exit menu item, and the "number of bins" edit box.

The File/Open callback must prompt the user for a file name and then read the data from the file. It must calculate and display the histogram and update the statistics text fields. Note that the data in the file must also be saved in the `handles` structure so that it will be available for recalculation if the user changes the number of bins in the historgram. The callback function to perform these steps is shown below:

```
function Open_Callback(hObject, eventdata, handles)
% hObject    handle to Open (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get file name
[filename,path] = uigetfile('*.dat','Load Data File');
if filename ~= 0

   % Read data
   x = textread([path filename],'%f');

   % Save in handles structure
   handles.x = x;
   guidata(gcbf, handles);

   % Create histogram
   hist(handles.x,handles.nbins);

   % Set axis labels
   xlabel('\bfValue');
   ylabel('\bfCount');

   % Calculate statistics
   ave = mean(x);
```

```
    med = median(x);
    sd  = std(x);
    n   = length(x);

    % Update fields
    set (handles.MeanData,'String',sprintf('%7.2f',ave));
    set (handles.MedianData,'String',sprintf('%7.2f',med));
    set (handles.StdDevData,'String',sprintf('%7.2f',sd));
    set (handles.TitleString,'String',['Histogram (N = 'int2str(n)')']);
end
```

The File/Exit callback is trivial. All it has to do is close the figure.

```
function Exit_Callback(hObject, eventdata, handles)
% hObject    handle to Exit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

close(gcbf);
```

The NBins callback must read a numeric input value, round it off to the nearest integer, display that integer in the Edit Box, and recalculate and display the histogram. Note that the number of bins must also be saved in the handles structure, so that it will be available for recalculation if the user loads a new data file. The callback function to perform these steps is shown below:

```
function NBins_Callback(hObject, eventdata, handles)
% hObject    handle to NBins (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get number of bins, round to integer, and update field
nbins = str2num(get(hObject,'String'));
nbins = round(nbins);
if nbins < 1
   nbins = 1;
end
set (handles.NBins,'String',int2str(nbins));

% Save in handles structure
handles.nbins = nbins;
guidata(gcbf, handles);

% Re-display data, if available
if handles.nbins > 0 & ~isempty(handles.x)

   % Create histogram
   hist(handles.x,handles.nbins);

end
```

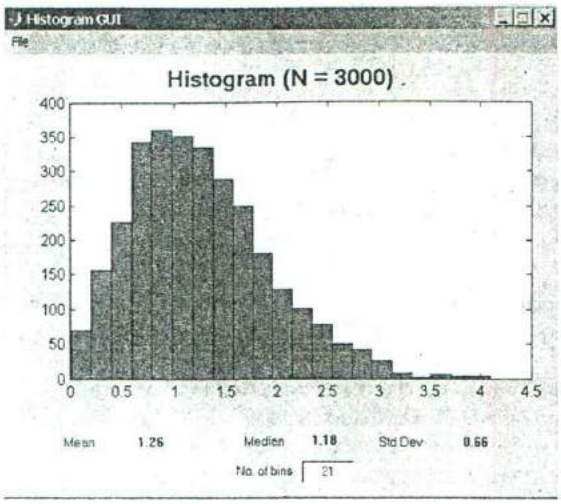


Figure 10.45 The GUI produced by program `histGUI`.

The final program is shown in Figure 10.45. Experiment with it on your own computer to verify that it behaves properly.

◀

10.9 Summary

In Chapter 10, we learned how to create MATLAB graphical user interfaces. The three fundamental parts of a GUI are components (`uicontrols`, `uimenus`, `uicontextmenus`, toolbars, and axes), containiers to contain them, and callbacks to implement actions in response to mouse clicks or keyboard inputs.

The standard GUI components created by `uicontrol` include text fields, edit boxes, pushbuttons, toggle buttons, checkboxes, radio buttons, popup menus, list boxes, and sliders. The standard GUI components created by `uimenu` and `uicontextmenu` are standard menus and context menus.

MATLAB containers consist of figures, panels, and button groups. Figures are created by the `figure` function. They are separate windows, complete with title bars, menus, and toolbars. Panels are created by the `uipanel` function. They are containers that reside within figures or other containers and do not have title bars, menus, or toolbars. Panels can contain `uicontrol` components and other panels or button groups, and those items will be laid out with respect to the

panel itself. If the panel is moved, all of its contents move with it. Button groups are created by the `uibuttongroup` function. They are special types of panels that control any radio buttons or toggle buttons contained within them to ensure that at most one of them can be on at any time.

Any of these components and containers can be placed on a figure using `guide` (the GUI Development Environment tool). Once the GUI layout has been completed, the user must edit the object properties with the Property Inspector and then write a callback function to implement the actions associated with each GUI object.

Dialog boxes are special figures used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. They are also used to select files or printer properties.

Dialog boxes may be modal or non-modal. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed, while a normal dialog box does not block access to other windows. Modal dialog boxes are typically used for warning and error messages that need urgent attention and cannot be ignored.

Menus can also be added to MATLAB GUIs. A menu allows a user to select actions without additional components appearing on the GUI display. They are useful for selecting less commonly used options without cluttering up the GUI with a lot of extra buttons. Menus are created with the Menu Editor, after which the programmer must write a callback function to implement the actions associated with each menu item. For each menu item, the user must use the Menu Editor to set at least the `Label` and `Tag` properties.

Accelerator keys and keyboard mnemonics can be used to speed the operation of windows.

Compiling MATLAB functions to pcode can speed the execution of a program. It also protects your investment in your source code by allowing you to distribute the program to others in the form of pcode files. They may be freely executed, but it is not easy for someone to reengineer the files and take your ideas.

MATLAB uicontrol components have several additional properties for specifying less common types of callbacks, including `CreateFcn`, `DeleteFcn`, `ButtonDownFcn`, and `KeyPressFcn`. MATLAB figures also have several properties for specifying types of callbacks, including `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`. These various callbacks allow a user to customize the appearance and response of this MATLAB GUIs to various user inputs.

Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB GUIs.

1. Use `guide` to lay out a new GUI, and use the Property Inspector to set the initial properties of each component, such as the text displayed on the

component, the color of the component, and the name of the callback function, if required.

2. After creating a GUI with `guide`, manually edit the resulting function to add comments describing its purpose and components and to implement the function of callbacks.
3. Store GUI application data in the `handles` structure so that it will automatically be available to any callback function.
4. If you modify any of the GUI application data in the `handles` structure, be sure to save the structure with a call to `guidata` before exiting the function where the modifications occurred.
5. Use dialog boxes to provide information or request input in GUI-based programs. If the information is urgent and should not be ignored, make the dialog boxes modal.
6. Define tool tips to provide users with helpful hints about the functions of your GUI components.
7. One a program is working properly, you may use the `pcode` command to pre-compile its M-files and increase the program's speed.

MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one. Also, refer to the summaries of graphical object properties in Tables 10.2, 10.3, 10.4, 10.6, and 10.7.

`axes`	Function to create a set of axes.
`dialog`	Create a generic dialog box.
`errordlg`	Display an error message.
`helpdlg`	Display a help message.
`findobj`	Find a GUI object by matching one or more of its properties.
`gcbf`	Get callback figure.
`gcbo`	Get callback object.
`guidata`	Save GUI application data in a figure.
`guihandles`	Get the `handles` structure from the application data stored in a figure.
`guide`	GUI Development Environment tool.
`inputdlg`	Dialog to get input data from the user.
`printdlg`	Print dialog box.
`questdlg`	Dialog box to ask a question.
`uibuttongroup`	Create a button group container.

`uicontrol`	Function to create a GUI object.
`uicontextmenu`	Function to create a context menu.
`uigetdir`	Dialog box to select a directory.
`uigetfile`	Dialog box to select an input file.
`uimenu`	Function to create a standard menu, or a menu item on either a standard menu or a context menu.
`uipanel`	Create a panel.
`uipushtool`	Create a pushbutton on a user-defined toolbar.
`uiputfile`	Dialog box to select an output file.
`uisetcolor`	Displays a color-selection dialog box.
`uisetfont`	Displays a font-selection dialog box.
`uitoggletool`	Create a toggle button on a user-defined toolbar.
`uitoolbar`	Create a user-defined toolbar.
`warndlg`	Displays a warning message.

10.10 Exercises

10.1 Explain the steps required to create a GUI in MATLAB.

10.2 What types of components can be used in MATLAB GUIs? What function creates them, and how do you select a particular component type?

10.3 What types of containers can be used in MATLAB GUIs? What function creates each of them?

10.4 How does a callback function work? How can a callback function locate the figures and objects that it needs to manipulate?

10.5 Create a GUI that uses a standard menu to select the background color displayed by the GUI. Include accelerator keys and keyboard mnemonics in the menu design. Design the GUI so that it defaults to a green background. [Note: You will need to use the figure's `OpeningFcn` to set the background color, since MATLAB defaults the figure color to the system default color.]

10.6 Create a GUI that uses a context menu to select the background color displayed by the GUI. Design the GUI so that it defaults to a yellow background.

10.7 Write a GUI program that plots the equation $y(x) = ax^2 + bx + c$. The program should include a set of axes for the plot and should include a panel containing GUI elements to input the values of a, b, c, and the minimum and maximum x to plot. A separate panel should contain controls to set the style, color, and thickness of the line being plotted. Include tool tips for each of your GUI elements.

10.8 Modify the GUI of Exercise 10.7 to include a menu. The menu should include two submenus to select the color and line style of the plotted line, with a check mark beside the currently selected menu choices. The menu should also include an "Exit" option. If the user selects the this option, the program should create a modal question dialog box asking "Are You Sure?" with the appropriate responses. Include accelerator keys and keyboard mnemonics in the menu design. [Note that the menu items duplicate some GUI elements; therefore, if a menu item is selected, the corresponding GUI elements must be updated, and vice versa.]

10.9 Modify the List Box example in Section 10.4 to allow for multiple selections in the list box. The text field should be expanded to multiple lines so that it can display a list of all selections whenever the "Select" button is clicked.

10.10 **Random Number Distributions** Create a GUI to display the distributions of different types of random numbers. The program should create the distributions by generating an array of 1,000,000 random values from a distribution and using function `hist` to create a histogram. Be sure to label the title and axes of the histogram properly.

The program should support uniform, Gaussian, and Rayleigh distributions, with the distribution selection made by a popup menu. In addition, it should have an edit box to allow the user to select the number of bins in the histogram. Make sure that the values entered in the edit box are legal (the number of bins must be a positive integer).

10.11 Modify the temperature conversion GUI of Example 10.1 to add a "thermometer." The thermometer should be set of rectangular axes with a red "fluid" level corresponding to the current temperature in degrees Celsius. The range of the thermometer should be 0°–100°C.

10.12 Modify the temperature conversion GUI of Exercise 10.11 to allow you to adjust the displayed temperature by clicking the mouse. (**Warning:** This exercise requires material not discussed in this chapter. Refer to the `CurrentPoint` property of `figure` objects in the on-line MATLAB documentation.)

10.13 Create a GUI that contains a title and four pushbuttons grouped within a panel. The pushbuttons should be labeled "Title Color," "Figure Color," "Panel Color," and "Title Font." If the "Title Color" button is selected, open a `uisetcolor` dialog box and change the title text to be in the selected color. If the "Figure Color" button is selected, open a `uisetcolor` dialog box and change the figure color and the title text background color to be the selected color. If the "Panel Color" button is selected, open a `uisetcolor` dialog box and change the panel background to be in the selected color. If the "Title Font" button is selected, open a `uisetfont` dialog box and change the title text to be in the selected font.

10.14 Create a GUI that contains a title and a button group. The button group will be titled "Style," and it should contain four radio buttons labeled "Plain," "Italic," "Bold," and "Bold Italic." Design the GUI so that the style in the currently selected radio button is applied to the title text.

10.15 **Least-Squares Fit** Create a GUI that can read an input data set from a file and perform a least-squares fit to the data. The data will be stored in a disk file in (x, y) format, with one x and one y value per line. Perform the least-squares fit with the MATLAB function `polyfit`, and plot both the original data and the least-squares fitted line. Include two menus: File and Edit. The File menu should include File/Open and File/Exit menu items, and the user should receive an "Are You Sure?" prompt before exiting. The Edit menu item should allow the user to customize the display, including line style, line color, and grid status.

10.16 Modify the GUI of the previous exercise to include an Edit/Preferences menu item that allows the user to suppress the "Are You Sure?" exit prompt.

10.17 Modify the GUI of the previous exercise to read and write an initialization file. The file should contain the line style, line color, grid choice (on/off), and exit prompt choice made by the user on previous runs. These choices should be automatically written out and saved when the program exits via the File/Exit menu item, and they should be read in and used whenever the program is started again.

APPENDIX A

ASCII Character Set

MATLAB strings use the ASCII character set, which consists of the 127 characters shown in the table below. The results of MATLAB string comparison operations depend on the *relative lexicographic positions* of the characters being compared. For example, the character 'a' in the ASCII character set is at position 97 in the table, while the character 'A' is at position 65. Therefore, the relational operator `'a' > 'A'` will return a 1 (true), since 97 > 65.

Each MATLAB character is stored in a 16-bit field, which means that MATLAB can support the entire Unicode character set.

The table shown below shows the ASCII character set, with the first two digits of the character number defined by the row and the third digit defined by the column. Thus, the letter `'R'` is on row 8 and column 2, so it is character 82 in the ASCII character set.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

APPENDIX B

Answers to Quizzes

This appendix contains the answers to all of the quizzes in the book.

Quiz 1.1, page 18

1. The MATLAB Command Window is the window where a user enters commands. A user can enter interactive commands at the command prompt (») in the Command Window, and they will be executed on the spot. The Command Window is also used to start M-files executing. The Edit/Debug Window is an editor used to create, modify, and debug M-files. The Figure Window is used to display MATLAB graphical output.
2. You can get help in MATLAB by:
   - Typing `help <command_name>` in the Command Window. This command will display information about a command or function in the Command Window.
   - Typing `lookfor <keyword>` in the Command Window. This command will display in the Command Window a list of all commands or functions containing the keyword in their first comment line.
   - Starting the Help Browser by typing `helpwin` or `helpdesk` in the Command Window, by selecting "Help" from the Start menu, or by clicking on the question mark icon ( ) on the desktop. The Help Browser contains an extensive hypertext-based description of all of the features in MATLAB, plus a complete copy of all

manuals on-line in HTML and Adobe PDF formats. It is the most comprehensive source of help in MATLAB.

3. A workspace is the collection of all the variables and arrays that can be used by MATLAB when a particular command, M-file, or function is executing. All commands executed in the Command Window (and all script files executed from the Command Window) share a common workspace, so they can all share variables. The contents of the workspace can be examined with the `whos` command, or graphically with the Workspace Browser.
4. To clear the contents of a workspace, type `clear` or `clear variables` in the Command Window.
5. The commands to perform this calculation are:

```
» t = 5;
» x0 = 10;
» v0 = 15;
» a = -9.81;
» x = x0 + v0 * t + 1/2 * a * t^2
x =
   -37.6250
```

6. The commands to perform this calculation are:

```
» x = 3;
» y = 4;
» res = x^2 * y^3 / (x - y)^2
res =
    576
```

Questions 7 and 8 are intended to get you to explore the features of MATLAB. There is no single "right" answer for them.

Quiz 2.1, page 30

1. An array is a collection of data values organized into rows and columns, and known by a single name. Individual data values within an array are accessed by including the name of the array followed by subscripts in parentheses that identify the row and column of the particular value. The term "vector" is usually used to describe an array with only one dimension, while the term "matrix" is usually used to describe an array with two or more dimensions.
2. (*a*) This is a 3×4 array; (*b*) `c(2, 3)` $= -0.6$; (*c*) The array elements whose value is 0.6 are `c(1, 4)`, `c(2, 1)`, and `c(3, 2)`.
3. (*a*) 1×3; (*b*) 3×1; (*c*) 3×3; (*d*) 3×2; (*e*) 3×3; (*f*) 4×3; (*g*) 4×1.

4. `w(2, 1)` = 2
5. `x(2, 1)` = $-20i$
6. `y(2, 1)` = 0
7. `v(3)` = 3

Quiz 2.2, page 39

1. (*a*) `c(2, :)` = [0.6 1.1 −0.6 3.1]

(*b*) `c(:, end)` = $\begin{bmatrix} 0.6 \\ 3.1 \\ 0.0 \end{bmatrix}$

(*c*) `c(1:2, 2:end)` = $\begin{bmatrix} -3.2 & 3.4 & 0.6 \\ 1.1 & -0.6 & 3.1 \end{bmatrix}$

(*d*) `c(6)` = 0.6

(*e*) `c(4, end)` = [−3.2 1.1 0.6 3.4 −0.6 5.5 0.6 3.1 0.0]

(*f*) `c(1:2, 2:end)` = $\begin{bmatrix} -3.2 & 3.4 & 0.6 \\ 1.1 & -0.6 & 3.1 \end{bmatrix}$

(*g*) `c([1 3], 2)` = $\begin{bmatrix} -3.2 \\ 0.6 \end{bmatrix}$

(*h*) `c([2 2], [3 3])` = $\begin{bmatrix} -0.6 & -0.6 \\ -0.6 & -0.6 \end{bmatrix}$

2. (*a*) `a` = $\begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$ (*b*) `a` = $\begin{bmatrix} 4 & 5 & 6 \\ 4 & 5 & 6 \\ 4 & 5 & 6 \end{bmatrix}$ (*c*) `a` = $\begin{bmatrix} 4 & 5 & 6 \\ 4 & 5 & 6 \end{bmatrix}$

3. (*a*) `a` = $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 3 \\ 0 & 0 & 1 \end{bmatrix}$ (*b*) `a` = $\begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 5 \\ 0 & 0 & 6 \end{bmatrix}$ (*c*) `a` = $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 9 & 7 & 8 \end{bmatrix}$

Quiz 2.3, page 45

1. The required command is "`format long e`".
2. (*a*) These statements get the radius of a circle from the user, and calculate and display the area of the circle. (*b*) These statements display the value of π as an integer, so they display the string: "`The value is 3!`".

3. The first statement outputs the value 12345.67 in exponential format; the second statement outputs the value in floating point format; the third statement outputs the value in general format; and the fourth statement outputs the value in floating point format in a field 12 characters wide, with four places after the decimal point. The results of these statements are:

```
value = 1.234567e+004
value = 12345.670000
value = 12345.7
value =  12345.6700
```

Quiz 2.4, page 52

1. (*a*) This operation is illegal. Array multiplication must be between arrays of the same shape, or between an array and a scalar. (*b*) Legal matrix multiplication: result = $\begin{bmatrix} 4 & 4 \\ 3 & 3 \end{bmatrix}$ (*c*) Legal array multiplication: result = $\begin{bmatrix} 2 & 1 \\ -2 & 4 \end{bmatrix}$ (*d*) This operation is illegal. The matrix multiplication `b * c` yields a 1×2 array, and `a` is a 2×2 array, so the addition is illegal. (*e*) This operation is illegal. The array multiplication `b .* c` is between two arrays of different sizes, so the multiplication is illegal.

2. This result can be found from the operation `x = A\B`: $x = \begin{bmatrix} -0.5 \\ 1.0 \\ -0.5 \end{bmatrix}$

Quiz 3.1, page 101

Expression	Result	Comment
1. `a > b`	1 (`logical` true)	
2. `b > d`	0 (`logical` false)	
3. `a > b & c > d`	0 (`logical` false)	

4. `a == b`	0 (`logical` false)	
5. `a & b > c`	0 (`logical` false)	
6. `~~b`	1 (`logical` true)	
7. `~(a > b)`	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ (`logical` array)	
8. `a > c && b > c`	Illegal	The `&&` and `||` operators only work between *scalar* operands.
9. `c <= d`	Illegal	The `<=` operator must be between arrays of the same size, or between an array and a scalar.
10. `logical(d)`	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ (`logical` array)	
11. `a * b > c`	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ (`logical` array)	The expression `a * b` is evalutated first, producing the `double` array $\begin{bmatrix} 2 & -4 \\ 0 & 20 \end{bmatrix}$, and the `logical` operation is evaluated second, producing the final answer.
12. `a * (b > c)`	$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ (`double` array)	The expression `b > c` produced the `logical` array $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, and multiplying that `logical` array by 2 converted the results back into a `double` array.
13. `a*b^2 > a*c`	0 (`logical` false)	
14. `d || b > a`	1 (`logical` true)	

15. `(d | b) > a`	0 (logical false)	
16. `isinf(a/b)`	0 (logical false)	
17. `isinf(a/c)`	1 (logical true)	
18. `a > b && ischar(d)`	1 (logical true)	
19. `isempty(c)`	0 (logical false)	
20. `(~a) & b`	0 (logical false)	
21. `(~a) + b`	−2 (double value)	~a is a logical 0. When added to b, the result is converted back to a double value.

Quiz 3.2, page 116

1.
```
if x >= 0
   sqrt_x = sqrt(x);
else
   disp('ERROR: x < 0');
   sqrt_x = 0;
end
```

2.
```
if abs(denominator) < 1.0E-300
   disp('Divide by 0 error.');
else
   fun = numerator / denominator;
   disp(fun)
end
```

3.
```
if distance <= 100
   cost = 0.50 * distance;
elseif distance <= 300
   cost = 50 + 0.30 * (distance - 100);
else
   cost = 110 + 0.20 * (distance - 300);
end
```

4. These statement are incorrect. For this structure to work, the second `if` statement would need to be an `elseif` statement.

5. These statement are legal. They will display the message "`Prepare to stop.`"
6. These statement will execute, but they will not do what the programmer intended. If the `temperature` is 150, these statements will print out "`Human body temperature exceeded.`" instead of "`Boiling point of water exceeded.`", because the `if` structure executes the first `true` condition and skips the rest. To get proper behavior, the order of these tests should be reversed.

Quiz 3.3, page 133

1.
```
x = 0:pi/10:2*pi;
x1 = cos(2*x);
y1 = sin(x);
plot(x1,y1,'-ro','LineWidth',2.0,'MarkerSize',6,...
     'MarkerEdgeColor','b','MarkerFaceColor','b')
```
3. `'\itf\rm(\itx\rm) = sin \theta cos 2\phi'`
4. `'\bfPlot of \Sigma \itx\rm\bf^{2} versus \itx'`
5. This string creates the characters: τ_m
6. This string creates the characters: $x_1^2 + x_2^2$ (units: m^2)
7. The backslash character is displayed using a double backslash (`'\\'`).

Quiz 4.1, page 174

1. 4 times
2. 0 times
3. 1 time
4. 2 times
5. 2 times
6. `ires = 10`
7. `ires = 55`
8. `ires = 25;`
9. `ires = 49;`
10. With loops and branches:
```
for ii = -6*pi:pi/10:6*pi
   if sin(ii) > 0
      res(ii) = sin(ii);
   else
      res(ii) = 0;
end
```

With vectorized code:

```
arr1 = sin(-6*pi:pi/10:6*pi);
res = zeros(size(arr1));
res(arr1>0) = arr1(arr1>0);
```

Quiz 5.1, page 222

1. Script files are collections of MATLAB statements that are stored in a file. Script files share the Command Window's workspace, so any variables that were defined before the script file starts are visible to the script file, and any variables created by the script file remain in the workspace after the script file finishes executing. A script file has no input arguments and returns no results, but script files can communicate with other script files through the data left behind in the workspace. In contrast, each MATLAB function runs in its own independent workspace. It receives input data through an input argument list, and returns results to the caller through an output argument list.
2. The `help` command displays all of the comment lines in a function until either the first blank line or the first executable statement is reached.
3. The H1 comment line is the first comment line in the file. This line is searched by and displayed by the `lookfor` command. It should always contain a one-line summary of the purpose of a function.
4. In the pass-by-value scheme, a *copy* of each input argument is passed from a caller to a function, instead of the original argument itself. This practice contributes to good program design because the input arguments may be freely modified in the function without causing unintended side effects in the caller.
5. A MATLAB function can have any number of arguments, and not all arguments need to be present each time the function is called. Function `nargin` is used to determine the number of input arguments actually present when a function is called, and function `nargout` is used to determine the number of output arguments actually present when a function is called.
6. This function call is incorrect. Function `test1` must be called with two input arguments. In this case, variable `y` will be undefined in function `test1`, and the function will abort.
7. This function call is correct. The function can be called with either one or two arguments.

Quiz 6.1, page 288

1. (*a*) `result = 1` (true), because the comparion is made between the real parts of the numbers (*b*) `result = 0` (false), because the absolute values of the two numbers are identical (*c*) `result = 25`
2. The function `plot(array)` plots the imaginary part of the array versus the real part of the array, with the real part on the *x* axis and the imaginary part on the *y* axis.
3. The vector can be converted using the `double` function.
4. These statements concatenate the two lines together, and variable res contains the string `'This is a test!This line, too.'`.
5. These statements are illegal—there is no function `strcati`.
6. These statements are illegal—the two strings must have the same number of columns, and these strings are of different lengths.
7. These statements are legal, producing the result `res = ['This is another test!'; 'This line, too.']`. Note that each line is now 21 characters long, with the line 2 padded out to that length.
8. These statements are legal, and the result `res = 1`, since the two strings are identical in their first 5 characters.
9. These statements are legal, and the result is `res = [4 7 13]`, since the letter "s" is at those locations in the string.
10. These statements are legal. Each space in the original string is replaced by an `'x'`, and the final string is `'Thisxisxaxtest!xx'`.
11. These statements are legal. The function isstrprop returns a 1 (`true`) for alphanumeric characters, and a 0 (`false`) for other characters. The result is

```
res =
     1     1     1     1     0     1     1     1     1     0     0     0
```

12. These statements are legal, with the result `res = 'ThiS IS a test!'`.
13. These statements are legal. The results are `11 = 9`, `12 = 9`, `13 = 18`, `14 = 6`, and `15 = 12`.
14. These statements are illegal—you must specify the number of characters to compare in the two strings when using function `strncmp`.

Quiz 7.1, page 352

1. A sparse array is a special type of array in which memory is only allocated for the non-zero elements in the array. Memory values are allocated for both the subscripts and the value of each element in a sparse array. By contrast, a memory location is allocated for every value in a full array, whether the value is 0 or not. Sparse arrays can be converted to full arrays using the `full` function, and full arrays can be converted to sparse arrays using the `sparse` function.
2. A cell array is an array of "pointers", each element of which can point to any type of MATLAB data. It differs from an ordinary array in that each element of a cell array can point to a different type of data, such as a numeric array, a string, another cell array, or a structure. Also, cell arrays use braces `{}` instead of parentheses `()` for selecting and displaying the contents of cells.
3. *Content indexing* involves placing braces `{}` around the cell subscripts, together with cell contents in ordinary notation. This type of indexing defines the contents of the data structure contained in a cell. *Cell indexing* involves placing braces `{}` around the data to be stored in a cell, together with cell subscripts in ordinary subscript notation. This type of indexing creates a data structure containing the specified data, and then assigns that data structure to a cell.
4. A structure is a data type in which each individual element is given a name. The individual elements of a structure are known as fields, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field, separated by a period. Structures differ from ordinary arrays and cell arrays in that ordinary arrays and cell array elements are addressed by subscript, while structure elements are addressed by name.
5. Function `varargin` appears as the last item in an input argument list, and it returns a cell array containing all of the actual arguments specified when the function is called, each in an individual element of a cell array. This function allows a MATLAB function to support any number of input arguments.
6. A function handle is a MATLAB data type that holds information to be used in referencing a function. A function handle is created using the `@` operator or the `str2func` function. A function is called using a function handle by naming the function handle followed by the arguments in parentheses. If the function has not arguments, the handle must be followed by empty parentheses `()`.

7. (*a*) `a(1, 1) = [3x3 double]`. The contents of cell array element `a(1, 1)` is a 3×3 double array, and this data structure is displayed.

(*b*) `a{1, 1}` = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. This statement displays the *value* of the data structure stored in element `a(1, 1)`.

(*c*) These statements are illegal, since you can not multiply a data structure by a value.

(*d*) These statements are legal, since you *can* multiply the contents of the data structure by a value. The result is $\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$.

(*e*) `a{2, 2}` = $\begin{bmatrix} -4 & -3 & -2 \\ -1 & 0 & 1 \\ 2 & 3 & 4 \end{bmatrix}$.

(*f*) This statement is legal. It initializes cell array element `a(2, 3)` to be a 2×1 double array containing the values $\begin{bmatrix} -17 \\ 17 \end{bmatrix}$.

(*g*) `a{2, 2}(2, 2) = 0`.

8. (*a*) `b(1).a - b(2).a` = $\begin{bmatrix} -3 & 1 & -1 \\ -2 & 0 & -2 \\ -3 & 3 & 5 \end{bmatrix}$

(*b*) `strncmp(b(1).b, b(2).b, 6) = 1`, since the two structure elements contain character strings that are identical in their first 6 characters.

(*c*) `mean(b(1).c) = 2`

(*d*) This statement is illegal, since you cannot treat individual elements of a structure array as though it were an array itself.

(*e*)
```
b = 1x2 struct array with fields:
    a
    b
    c
```

(*f*)
```
b(1).('b') = 'Element 1'
```

(*g*)
```
b(1) =
    a:  [3x3 double]
    b:  'Element 1'
    c:  [1 2 3]
```

9. (*a*) The result of this function is the name of function whose handle is passed as a calling argument:

```
» myfun(@cosh)
ans =
Element 1
```

Quiz 8.1, page 372

1. The `textread` function is designed to read ASCII files that are formatted into columns of data, where each column can be of a different type. This command is *very* useful for importing tables of data printed out by other applications, since it can handle data of mixed types within a single file.
2. MAT files are relatively efficient users of disk space, and they store the full precision of every variable—no precision is lost due to conversion to and from ASCII format. (If compression is used, MAT files take up even less space.) In addition, MAT files preserve all of the information about each variable in the workspace, including its class, name, and whether or not it is global. A disadvantage of MAT files is that they are unique to MATLAB, and cannot be used to share data with other programs.
3. Function `fopen` is used to open files, and function `fclose` is used to close files. On PCs (but not on Linux or Unix computers), there is a difference between the format of a text file and a binary file. In order to open files in text mode on a PC, a `'t'` must be appended to the permission string in the `fopen` function.
4. `fid = fopen('myinput.dat','at')`
5.
```
fid = fopen('input.dat','r');
if fid < 0;
   disp('File input.dat does not exist.');
end
```
6. These statements are incorrect. They open a file as a text file, but then read the data in binary format. (Function `fscanf` should be used to read text data, as we see later in this chapter.)
7. These statements are correct. They create a 10-element array `x`, open a binary output file `file1`, write the array to the file, and close the file. Next, they open the file again for reading, and read the data into array `array` in a `[2 Inf]` format. The resulting contents of the array are $\begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 & 10 \end{bmatrix}$.

Quiz 8.2, page 388

1. Formatted I/O operations produce formatted files. Aformatted file contains recognizable characters, numbers, *etc.* stored as ASCII text. Formatted files have the advantages that we can readily see what sort of data they contain, and it is easy to exchange data between different types of programs using them. However, formatted I/O operations take longer to read and write, and formatted files take up more space than unformatted files. Unformatted I/O operations copy the information from a computer's memory directly to the disk file with no conversions at all. These operations are much faster than formatted I/O operations because there is no conversion. In addition, the data occupies a much smaller amount of disk space. However, unformatted data cannot be examined and interpreted directly by humans.
2. Formatted I/O should be used whenever we need to exchange data between MATLAB and other programs, or when a person needs to be able to examine and/or modify the data in the file. Otherwise, unformatted I/O should be used.
3.
```
fprintf('Table of Cosines and Sines\n\n');
fprintf(' theta   cos(theta)   sin(theta)\n');
fprintf(' =====   ==========    ==========\n');
for ii = 0:0.1:1
   theta = pi * ii;
   fprintf('%7.4f %11.5f %11.5f\n',
            theta,cos(theta),sin(theta));
end
```
4. These statements are incorrect. The `%s` descriptor must correspond to a character string in the output list.
5. These statements are technically correct, but the results are undesirable. It is possible to mix binary and formatted data in a single file the way that these statements do, but the file is then very hard to use for any purpose. Normally, binary data and formatted data should be written to separate files.

Quiz 10.1, page 494

1. The types of graphical components discussed in this chapter are listed below, together with their purposes.

Table B.1 GUI Components Discussed in Chapter 10

Component	Created By	Description
	Graphical Controls	
Pushbutton	`uicontrol`	A graphical component that implements a pushbutton. It triggers a callback when clicked with a mouse.
Toggle Button	`uicontrol`	A graphical component that implements a toggle button. A toggle button is either "on" or "off", and it changes state each time that it is clicked. Each mouse button click also triggers a callback.
Radio Button	`uicontrol`	A radio button is a type of toggle button that appears as a small circle with a dot in the middle when it is "on". Groups of radio buttons are used to implement mutually exclusive choices. Each mouse click on a radio button triggers a callback.
Check Box	`uicontrol`	A checkbox is a type of toggle button that appears as a small square with a check mark in it when it is "on". Each mouse click on a check box triggers a callback.
Edit Box	`uicontrol`	An edit box displays a text string, and allows the user to modify the information displayed. A callback is triggered when the user presses the `Enter` key.
List Box	`uicontrol`	A list box is a graphical control that displays a series of text strings. A user may select one of the text strings by single- or double-clicking on them. A callback is triggered when the user selects a string.
Popup Menus	`uicontrol`	A popup menu is a graphical control that displays a series of text strings in response to a mouse click. When the popup menu is not clicked on, only the currently-selected string is visible.
Slider	`uicontrol`	A slider is a graphical control to adjust a value in a smooth, continuous fashion by dragging the control with a mouse. Each slider change triggers a callback.
	Static Elements	
Frame	`uicontrol`	Creates a frame, which is a rectangular box within a figure. Frames are used to group sets of controls together. Frames never trigger callbacks. (This is a deprecated component, which should not be used in new GUIs.)
Text Field	`uicontrol`	Creates a label, which is a text string located at a point on the figure. Text fields never trigger callbacks.
	Menus, Toolbars, Axes	
Menu Items	`uimenu`	Creates a menu item. Menu items trigger a callback when a mouse button is released over them.

(continued)

Table B.1 (*continued*)

Component	Created By	Description
Context Menus	`uicontextmenu`	Creates a context menu, which is a menu that appears over a graphical object when a user right-clicks the mouse on that object.
Toolbar	`uitoolbar`	Creates a toolbar, which is a bar across the top of the figure containing quick-access buttons.
Toolbar Pushbutton	`uipushtool`	Creates a pushbutton to go in a toolbar.
Toolbar Toggle Button	`uitoggletool`	Creates a toggle button to go in a toolbar.
Axes	`axes`	Creates a new set of axes to display data on. Axes never trigger callbacks.

2. The types of containerss discussed in this chapter are listed below, together with their differences.

Table B.2 GUI Components Discussed in Chapter 10

Component	Created By	Description
		Containers
Figure	`uicontrol`	Creates a figure, which is container that can hold components and other containers. Figures are separate windows that have title bars, and can have menus.
Panel	`uipanel`	Creates a panel, which is container that can hold components and other containers. Unlike figures, panels do not have title bars or menus. Panels can be placed inside figures or other panels.
Button Group	`uibuttongroup`	Creates a button group, which is a special kind of panel. Button groups automatically manage groups of radio buttons or toggle buttons to ensure that only one item of the group is on at any given time.

3. A callback function is a function that is executed whenever an action (mouse click, keyboard input, etc.) occurs on a specific GUI component. They are used to perform an action when a user clicks on or types in a GUI component. Callback functions are specified by the `'Callback'` property in a `uicontrol`, `uimenu`, `uicontextmenu`, `uipushtool`, or `uitoggletool`. When a new GUI is created, the callbacks are set automatically by `guide` to be `xxx_Callback`, where `xxx` is the value of the `Tag` property of the corresponding GUI component.

4. The basic steps required to create a MATLAB GUI are:
   1. Decide what elements are required for the GUI, and what the function of each element will be. Make a rough layout of the components by hand on a piece of paper.
   2. Use a MATLAB tool called `guide` (GUI Development Environment) to lay out the components on a figure. The size of the figure, and the alignment and spacing of components on the figure, can be adjusted using the tools built into guide.
   3. Use a MATLAB tool called the Property Inspector (built into `guide`) to give each component a name (a "tag"), and to set the characteristics of each component, such as its color, the text it displays, etc.
   4. Save the figure to a file. When the figure is saved, two files will be created on disk with the same name but different extents. The `fig` file contains the actual GUI that you have created, and the M-file contains the code to load the figure, and also skeleton callbacks for each GUI element.
   5. Write code to implement the behavior associated with each callback function.
5. The `handles` data structure is a structure containing the handles of all components within a figure. Each structure element has the name of a component, and the value of the component's handle. This structure is passed to every callback function, allowing each function to have access to every component in the figure.
6. Application data can be saved in a GUI by adding it to the handles structure, and saving that structure after it is modified using function `guidata`. Since the `handles` structure is automatically passed to every callback function, and additional data added to the structure will be available to any callback function in the GUI. (Each function that modifies the handles structure must be sure to save the modified version with a call to `guidata` before the function exits.)
7. A graphical object can be made invisible by setting its `'Visible'` property to `'off'`. A graphical object can be disabled so that it will not respond to mouse clicks or keyboard input by setting its `'Enable'` property to `'off'`.
8. Pushbuttons, toggle buttons, radio buttons, check boxes, list boxes, popup menus, and sliders all respond to mouse clicks. Edit boxes respond to keyboard inputs.
9. A dialog box is a special type of figure that is used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. Dialog

boxes can be created by any of the functions listed in Table 10.5, including `errordlg`, `warndlg`, `inputdlg`, `uigetfile`, etc.

10. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed, while a normal dialog box does not block access to other windows.

11. A standard menu is tied to a menu bar running across the top of a figure, while a context menu can be attached to any GUI component. Standard menus are activated by normal mouse clicks on the menu bar, while context menus are activated by mouse right-clicks over the associated GUI component. Menus are built out of `uimenu` components. Context menus are built out of both `uicontextmenu` and `uimenu` components.

12. Accelerator keys are keys that may be typed on the keyboard to cause a menu item to be selected. Keyboard mnemonic keys are CTRL+key combinations that cause a menu item to be executed. The principal difference between accelerator keys and keyboard mnemonics is that accelerator keys only work to select a menu item if a menu has already been opened, while keyboard mnemonics can trigger an action even if a menu has not been opened.