CHAPTER 9

Data Transfer

KEY TERMS

file input and output
lower-level file I/O functions
file types
open the file
close the file
permission
file identifier
standard input
standard output
standard error
end of the file
Application Programming Interface (API)
RESTful API calls
endpoint
query parameters

CONTENTS

This chapter extends the input and output concepts (I/O) that were introduced in Chapter 3. In that chapter, we saw how to read values entered by the user using the **input** function and also the output functions **disp** and **fprintf** that display information in windows on the screen.

For ***file input and output*** (file I/O), we used the **load** and **save** functions that can read from a data file into a matrix, and write from a matrix to a data file. If the data to be written or file to be read are not in a simple matrix format, ***lower-level file I/O functions*** must be used.

The **save** and **load** functions can also be used to store MATLAB® variables, both their name and their contents, into special MATLAB binary files called MAT-files and to read those variables into the base workspace.

The MATLAB software has functions that can read and write data from different ***file types*** such as spreadsheets. For example, it can read from and write to Excel spreadsheets that have filename extensions such as .xls or .xlsx. There is also an Import Tool, which allows one to import data from a variety of file formats.

In this chapter, we will introduce some functions that work with different file types, as well as the programmatic methods using some of the lower-level file input and output functions. Additionally, MATLAB has functions that allow you to access data from websites; these functions will be introduced.

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00009-X

9.1 USING MAT-FILES FOR VARIABLES

In addition to the functions that manipulate data files, MATLAB has functions that allow reading variables from and saving variables to files. These files are called MAT-files (because the extension on the file name is .mat), and they store the names and contents of variables. Variables can be written to MAT-files, appended to them, and read from them.

9.1.1 Writing Variables to a File

Note
MAT-files are very different from the data files that we have worked with so far. Rather than just storing data, MAT-files store the variable names in addition to their values. These files are typically used only within MATLAB; they are not used to share data with other programs.

The **save** command can be used to write variables to a MAT-file or to append variables to a MAT-file. By default, the **save** function writes to a MAT-file. It can either save the entire current workspace (all variables that have been created) or a subset of the workspace (including, e.g., just one variable). The **save** function will save the MAT-file in the Current Folder, so it is important to set that correctly first.

To save all workspace variables in a file, the command is:

```
save filename
```

The '.mat' extension is added to the filename automatically. The contents of the file can be displayed using **who** with the '-file' qualifier:

```
who –file filename
```

For example, in the following session in the Command Window, three variables are created; these are then displayed using **who**. Then, the variables are saved to a file named "sess1.mat". The **who** function is then used to display the variables stored in that file.

```
>> mymat = rand(3,5);
>> x = 1:6;
>> y = x.^2;
>> who
Your variables are:
mymat  x    y
>> save sess1
>> who -file sess1
Your variables are:
mymat  x    y
```

To save just one variable to a file, the format is

```
save filename variablename
```

For example, just the matrix variable *mymat* is saved to a file called *sess2*:

```
>> save sess2 mymat
>> who -file sess2
Your variables are:
mymat
```

9.1.2 Appending Variables to a MAT-File

Appending to a file adds to what has already been saved in a file and is accomplished using the –append option. For example, assuming that the variable *mymat* has already been stored in the file "sess2.mat" as just shown, this would append the variable *x* to the file:

```
>> save -append sess2 x
>> who -file sess2
Your variables are:
mymat  x
```

Without specifying variable(s), just **save –append** would add all variables from the base workspace to the file. When this happens, if the variable is not in the file, it is appended. If there is a variable with the same name in the file, it is replaced by the current value from the base workspace.

9.1.3 Reading From a MAT-File

The **load** function can be used to read from different types of files. As with the **save** function, by default the file will be assumed to be a MAT-file, and **load** can load all variables from the file or only a subset. For example, in a new Command Window session in which no variables have yet been created, the **load** function could load from the files created in the previous section:

```
>> who
>> load sess2
>> who
Your variables are:
mymat  x
```

A subset of the variables in a file can be loaded by specifying them in the form:

```
load filename variable list
```

9.2 WRITING AND READING SPREADSHEET FILES

MATLAB has functions **xlswrite** and **xlsread** that will write to and read from Excel spreadsheet files that have extensions such as '.xls'. (Note that this works under Windows environments provided that Excel is loaded. Under other

environments, problems may be encountered if Excel cannot be loaded as a COM server. In this case, xlswrite will write the file in comma separated value format.) For example, the following will create a 5×3 matrix of random integers, and then write it to a spreadsheet file "ranexcel.xls" that has five rows and three columns:

```
>> ranmat = randi(100,5,3)
ranmat =
     96    77    62
     24    46    80
     61     2    93
     49    83    74
     90    45    18
>> xlswrite('ranexcel',ranmat)
```

The **xlsread** function will read from a spreadsheet file. For example, use the following to read from the file "ranexcel.xls":

```
>> ssnums = xlsread('ranexcel')
ssnums =
     96    77    62
     24    46    80
     61     2    93
     49    83    74
     90    45    18
```

In both cases, the '.xls' extension on the file name is the default, so it can be omitted.

These are shown in their most basic forms, when the matrix and/or spreadsheet contains just numbers and the entire spreadsheet is read or matrix is written. There are many qualifiers that can be used for these functions, however. For example, the following would read from the spreadsheet file "texttest.xls" that contains:

a	123	Cindy
b	333	Suzanne
c	432	David
d	987	Burt

```
>> [nums, txt] = xlsread('texttest.xls')
nums =
    123
    333
    432
    987
```

```
txt =
  4×3 cell array
     {'a'}    {0×0 char}    {'Cindy'   }
     {'b'}    {0×0 char}    {'Suzanne'}
     {'c'}    {0×0 char}    {'David'   }
     {'d'}    {0×0 char}    {'Burt'    }
```

This reads the numbers into a **double** vector variable *nums* and the text into a cell array *txt* (the **xlsread** function always returns the numbers first and then the text). The cell array is 4×3. It has three columns as the file had three columns, but as the middle column had numbers (which were extracted and stored in the vector *nums*), the middle column in the cell array *txt* consists of empty character vectors.

A loop could then be used to echo print the values from the spreadsheet in the original format:

```
>> for i = 1:length(nums)
       fprintf('%c %d %s\n', txt{i,1}, ...
          nums(i), txt{i,3})
   end
a 123 Cindy
b 333 Suzanne
c 432 David
d 987 Burt
```

These are just examples; MATLAB has many other functions that read from and write to different file formats.

9.3 LOWER-LEVEL FILE I/O FUNCTIONS

When reading from a data file, the **load** function works as long as the data in the file are "regular"—in other words, the same kind of data on every line and in the same format on every line—so that they can be read into a matrix. However, data files are not always set up in this manner. When it is not possible to use **load**, MATLAB has what are called lower-level file input functions that can be used. The file must be opened first, which involves finding or creating the file and positioning an indicator at the beginning of the file. This indicator then moves through the file as it is being read from. When the reading has been completed, the file must be closed.

Similarly, the **save** function can write or append matrices to a file, but if the output is not a simple matrix, there are lower-level functions that write to files. Again, the file must be opened first and closed when the writing has been completed.

In general, the steps involved are:

- ***open the file***
- read from the file, write to the file, or append to the file
- ***close the file***

First, the steps involved in opening and closing the file will be described. Several functions that perform the middle step of reading from or writing to the file will be described subsequently.

9.3.1 Opening and Closing a File

Files are opened with the **fopen** function. By default, the **fopen** function opens a file for reading. If another mode is desired, a ***permission*** is used to specify which, for example, writing or appending. The **fopen** function returns −1 if it is not successful in opening the file or an integer value that becomes the ***file identifier*** if it is successful. This file identifier is then used to refer to the file when calling other file I/O functions. The general form is

```
fid = fopen('filename', 'permission');
```

where *fid* is a variable that stores the file identifier (it can be named anything) and the permission includes:

r reading (this is the default)
w writing
a appending

After the **fopen** is attempted, the value returned should be tested to make sure that the file was opened successfully. For example, if attempting to open for reading and the file does not exist, the **fopen** will not be successful. As the **fopen** function returns −1 if the file was not found, this can be tested to decide whether to print an error message or to carry on and use the file. For example, if it is desired to read from a file "samp.dat":

```
fid = fopen('samp.dat');
if fid == -1
   disp('File open not successful')
else
   % Carry on and use the file!
end
```

The **fopen** function also returns an error message if it is not successful; this can be stored and displayed. Also, when the first file is opened in a MATLAB session, it will have a file identifier value of 3, because MATLAB assigns three default identifiers (0, 1, and 2) for the ***standard input***, ***standard output***, and ***standard error***. This can be seen if the output from an **fopen** is not suppressed.

```
>> [fid, msg] = fopen('sample.dat')
fid =
    -1
msg =
No such file or directory
>> if fid == -1
      error(msg)
   else
      % Carry on and use the file!
   end
No such file or directory
>> [fid, msg] = fopen('samp.dat')
fid =
     3
msg =
  0×0 empty char array
```

In the last case, there was no error so the message is an empty character vector.

Files should be closed when the program has finished reading from or writing or appending to them. The function that accomplishes this is the **fclose** function, which returns 0 if the file close was successful or −1 if not. Individual files can be closed by specifying the file identifier or, if more than one file is open, all open files can be closed by passing the string "all" (or character vector 'all') to the **fclose** function. The general forms are:

```
closeresult = fclose(fid);
closeresult = fclose("all")
```

The result from the **fclose** function should also be checked with an **if-else** statement to make sure it was successful, and a message should be printed (if the close was not successful, that might mean that the file was corrupted and the user would want to know that). So, the outline of the code will be:

```
fid = fopen('filename', 'permission' );
if fid == -1
   disp('File open not successful')
else

   % do something with the file!

   closeresult = fclose(fid);

   if closeresult == 0
      disp('File close successful')
   else
      disp('File close not successful')
     end
end
```

9.3.2 Reading From Files Using fgetl

There are several lower-level functions that read from files. The **fgetl** and **fgets** functions read text from a file one line at a time; the difference is that the **fgets** keeps the newline character if there is one at the end of the line, whereas the **fgetl** function gets rid of it. Both of these functions require first opening the file, and then closing it when finished. As the **fgetl** and **fgets** functions read one line at a time, these functions are typically inside some form of a loop.

We will concentrate on the **fgetl** function, which reads character vectors from a file one line at a time. Other input functions will be covered in Section 9.3.4. The **fgetl** function affords more control over how the data are read than other input functions. The **fgetl** function reads one line of data from a file into a character vector; text manipulation functions can then be used to manipulate the data. As **fgetl** only reads one line, it is normally placed in a loop that keeps going until the ***end of the file*** is reached. The function **feof** returns **logical true** if the end of the file has been reached. The function call **feof(fid)** would return **logical true** if the end of the file has been reached for the file identified by *fid*, or **logical false** if not. The **fgetl** function returns a character vector or −1 if no more data are found in the file. It is actually more robust in MATLAB to check the value returned by **fgetl** than to use **feof**.

A general algorithm for reading from a file into character vectors would be:

- Attempt to open the file
  - Check to ensure the file open was successful.
- If opened, loop until there is no more data
  - For each line in the file:
    - read it into a character vector
    - manipulate the data
- Attempt to close the file
  - Check to make sure the file close was successful.

The following is the generic code to accomplish these tasks:

```
fid = fopen('filename');
if fid == -1
   disp('File open not successful')
else
   % Attempt to read a line and check
   aline = fgetl(fid);
   while aline != -1
       % Use text functions to extract numbers,
       %   character vectors, etc. from the line
       % Do something with the data!
       % Attempt to read another line
```

```
        aline = fgetl(fid);
    end
    closeresult = fclose(fid);
    if closeresult == 0
        disp('File close successful')
      else
        disp('File close not successful')
      end
end
```

The permission could be included in the call to the **fopen** function. For example:

```
fid = fopen('filename', 'r');
```

but the 'r' is not necessary as reading is the default.

For example, assume that there is a data file "subjexp.dat", which has on each line a number followed by a space followed by a character code. The **type** command can be used to display the contents of this file (as the file does not have the default extension .m, the extension on the file name must be included).

```
>> type subjexp.dat
5.3 a
2.2 b
3.3 a
4.4 a
1.1 b
```

The **load** function would not be able to read this into a matrix as it contains both numbers and text. Instead, the **fgetl** function can be used to read each line as a character vector, and then text functions are used to separate the numbers and characters. For example, the following just reads each line and prints the number with two decimal places and then the rest of the character vector:

fileex.m

```
% Reads from a file one line at a time using fgetl
% Each line has a number and a character
% The script separates and prints them

% Open the file and check for success
fid = fopen('subjexp.dat');
if fid == -1
   disp('File open not successful')
else
   aline = fgetl(fid);
   while aline ~= -1
        % Separate each line into the number and character
        % code and convert to a number before printing
```

```
        [num, charcode] = strtok(aline);
        fprintf('%.2f %s\n', str2double(num), charcode)
        aline = fgetl(fid);
    end

    % Check the file close for success
    closeresult = fclose(fid);
    if closeresult == 0
        disp('File close successful')
    else
        disp('File close not successful')
    end
end
```

The following is an example of executing this script:

```
>> fileex
5.30 a
2.20 b
3.30 a
4.40 a
1.10 b
File close successful
```

In this example, every time the loop action is executed, the **fgetl** function reads one line into a character vector. The function **strtok** is then used to store the number and the character in separate variables, both of which are character vectors (the second variable actually stores the blank space and the letter). If it is desired to perform calculations using the number, the function **str2double** would be used to convert the number stored in the variable into a **double** value as shown.

PRACTICE 9.1

Modify the script *fileex* to sum the numbers from the file. Create your own file in this format first.

9.3.3 Writing and Appending to Files

There are several lower-level functions that can write to files. Like the other low-level functions, the file must be opened first for writing (or appending) and should be closed once the writing has been completed.

We will concentrate on the **fprintf** function, which can be used to write to a file and also to append to a file. To write one line at a time to a file, the **fprintf** function can be used. We have, of course, been using **fprintf** to write to the screen. The screen is the default output device, so if a file identifier is not specified, the

output goes to the screen; otherwise, it goes to the specified file. The default file identifier number is 1 for the screen. The general form is:

```
fprintf(fid, 'format', variable(s));
```

The **fprintf** function actually returns the number of bytes that was written to the file, so if it is not desired to see that number, the output should be suppressed with a semicolon as shown here.
The following is an example of writing to a file named "tryit.txt":

Note
When writing to the screen, the value returned by **fprintf** is not seen, but could be stored in a variable.

```
>> fid = fopen('tryit.txt', 'w');
>> for i = 1:3
       fprintf(fid,'The loop variable is %d\n', i);
   end
>> fclose(fid);
```

The permission in the call to the **fopen** function specifies that the file is opened for writing to it. Just like when reading from a file, the results from **fopen** and **fclose** should really be checked to make sure they were successful. The **fopen** function attempts to open the file for writing. If the file already exists, the contents are erased, so it is as if the file had not existed. If the file does not currently exist (which would be the norm), a new file is created. The **fopen** could fail, for example, if there isn't space to create this new file.

To see what was written to the file, we could then open it (for reading) and loop to read each line using **fgetl**:

```
>> fid = fopen('tryit.txt');
>> aline = fgetl(fid);
>> while aline ~= -1
       disp(aline)
       aline = fgetl(fid);
   end
The loop variable is 1
The loop variable is 2
The loop variable is 3
>> fclose(fid);
```

Of course, we could also just display the contents using **type**.

Here is another example in which a matrix is written to a file. First, a 2×4 matrix is created and then it is written to a file using the format specifier '%d %d\n', which means that each column from the matrix will be written as a separate line in the file.

```
>> mat = [20 14 19 12; 8 12 17 5]
mat =

    20    14    19    12
     8    12    17     5

>> fid = fopen('randmat.dat','w');
>> fprintf(fid,'%d %d\n',mat);
>> fclose(fid);
```

As this is a matrix, the **load** function can be used to read it in.

```
>> load randmat.dat
>> randmat
randmat =
    20     8
    14    12
    19    17
    12     5

>> randmat'
ans =
    20    14    19    12
     8    12    17     5
```

Transposing the matrix will display in the form of the original matrix. If this is desired to begin with, the matrix variable *mat* can be transposed before using **fprintf** to write to the file. (Of course, it would be much simpler in this case to just use **save** instead!)

PRACTICE 9.2

Create a 3×5 matrix of random integers, each in the range from 1 to 100. Write the sum of each row to a file called "myrandsums.dat" using **fprintf**. Confirm that the file was created correctly.

The **fprintf** function can also be used to append to an existing file. The permission is 'a', so the general form of the **fopen** would be:

```
fid = fopen('filename', 'a');
```

Then, using **fprintf** (typically in a loop), we would write to the file starting at the end of the file. The file would then be closed using **fclose**. What is written to the end of the file doesn't have to be in the same format as what is already in the file when appending.

9.3.4 Alternate File Input Functions

The function **fscanf** reads formatted data into a matrix, using conversion formats such as %d for integers, %s for strings, and %f for floats (**double** values).

The **textscan** function reads text data from a file and stores the data in a cell array; it also uses conversion formats. The **fscanf** and **textscan** functions can read the entire data file into one data structure. In terms of level, these two functions are somewhat in between the **load** function and the lower-level functions, such as **fgetl**. The file must be opened using **fopen** first and should be closed using **fclose** after the data has been read. However, no loop is required; these functions will read in the entire file automatically into a data structure.

Instead of using the **fgetl** function to read one line at a time, once a file has been opened the **fscanf** function can be used to read from this file directly into a matrix. However, the matrix must be manipulated somewhat to get it back into the original form from the file. The format of using the function is:

```
mat = fscanf(fid, 'format', [dimensions])
```

The **fscanf** reads into the matrix variable *mat* columnwise from the file identified by *fid*. The 'format' includes conversion characters much like those used in the **fprintf** function. The 'format' specifies the format of every line in the file, which means that the lines must be formatted consistently. The dimensions specify the desired dimensions of *mat*; if the number of values in the file is not known, **inf** can be used for the second dimension.

For example, the following would read in from the file *subjexp.dat*; each line contains a number, followed by a space, and then a character.

```
>> type subjexp.dat
5.3 a
2.2 b
3.3 a
4.4 a
1.1 b
>> fid = fopen('subjexp.dat');
>> mat = fscanf(fid,'%f %c',[2, inf])
mat =
    5.3000    2.2000    3.3000    4.4000    1.1000
   97.0000   98.0000   97.0000   97.0000   98.0000
>> fclose(fid);
```

The **fopen** opens the file for reading. The **fscanf** then reads from each line one double and one character and places each pair in separate columns in the matrix (in other words, every line in the file becomes a column in the matrix). Note that the space in the format specifier is important: `'%f %c'` specifies that there is a float, a space, and a character. The dimensions specify that the matrix is to have two rows by however many columns are necessary (equal to the number of lines in the file). As matrices store values that are all of the same type, the characters are stored as their ASCII equivalents in the character encoding (e.g., 'a' is 97).

Once this matrix has been created, it may be more useful to separate the rows into vector variables and to convert the second back to characters, which can be accomplished as follows:

```
>> nums = mat(1,:);
>> charcodes = char(mat(2,:))
charcodes =
abaab
```

Of course, the results from **fopen** and **fclose** should be checked but were omitted here for simplicity.

PRACTICE 9.3

Write a script to read in this file using **fscanf**, and sum the numbers.

QUICK QUESTION!

Instead of using the dimensions [2, **inf**] in the **fscanf** function, could we use
[**inf** ,2]?

Answer: No, [**inf**, 2] would not work. Because **fscanf** reads each row from the file into a column in the matrix, the number of rows in the resulting matrix is known but the number of columns is not.

QUICK QUESTION!

Why is the space in the conversion specifier `'%f %c'` important? Would the following also work?

```
>> mat = fscanf(fid,'%f%c',[2, inf])
```

Answer: No, that would not work. The conversion specifier `'%f %c'` specifies that there is a real number, then a space, then a character. Without the space in the conversion specifier, it would specify a real number immediately followed by a character (which would be the space in the file). Then, the next time it would be attempting to read the next real number, the file position indicator would, however, be pointing to the character on the first line; the error would cause the **fscanf** function to halt. The end result follows:

```
>> fid = fopen('subjexp.dat');
>> mat = fscanf(fid,'%f%c',[2, inf])
mat =
    5.3000
   32.0000
```

The 32 is the numerical equivalent of the space character ' ', as seen here.

```
>> double(' ')
ans =
    32
```

Another option for reading from a file is to use the **textscan** function. The **textscan** function reads text data from a file and stores the data in column vectors in a cell array. The **textscan** function is called, in its simplest form, as

```
cellarray = textscan(fid, 'format');
```

where the 'format' includes conversion characters much like those used in the **fprintf** function. The 'format' essentially describes the format of columns in the data file, which will then be read into column vectors. For example, to read the file 'subjexp.dat' we could do the following (again, for simplicity, omitting the error-check of **fopen** and **fclose**):

```
>> fid = fopen('subjexp.dat');
>> subjdata = textscan(fid,'%f %c');
>> fclose(fid);
```

The format specifier `'%f %c'` specifies that on each line there is a **double** value followed by a space followed by a character. This creates a 1×2 cell array variable called *subjdata*. The first element in this cell array is a column vector of doubles (the first column from the file); the second element is a column vector of characters (the second column from the file), as shown here:

```
>> subjdata
subjdata =
  1×2 cell array
    {5×1 double}  {5×1 char}
>> subjdata{1}
ans =
    5.3000
    2.2000
    3.3000
    4.4000
    1.1000
>> subjdata{2}
ans =
  5×1 char array
    'a'
    'b'
    'a'
    'a'
    'b'
```

To refer to individual values from the vector, it is necessary to index into the cell array using curly braces and then index into the vector using parentheses. For example, to refer to the third number in the first element of the cell array:

```
>> subjdata{1}(3)
ans =
    3.3000
```

A script that reads in these data and echo prints it is shown here:

textscanex.m

```
% Reads data from a file using textscan
fid = fopen('subjexp.dat');
if fid == -1
   disp('File open not successful')
else
   % Reads numbers and characters into separate elements
   % in a cell array
   subjdata = textscan(fid,'%f %c');
   len = length(subjdata{1});
   for i = 1:len
       fprintf('%.1f %c\n',subjdata{1}(i),subjdata{2}(i))
   end

   closeresult = fclose(fid);
   if closeresult == 0
       disp('File close successful')
   else
       disp('File close not successful')
   end
end
```

Executing this script produces the following results:

```
>> textscanex
5.3 a
2.2 b
3.3 a
4.4 a
1.1 b
File close successful
```

PRACTICE 9.4

Modify the script *textscanex* to calculate the average of the column of numbers.

9.3.4.1 Comparison of Input File Functions

To compare the use of these input file functions, consider the example of a file called "xypoints.dat" that stores the x and y coordinates of some data points in the following format:

```
>> type xypoints.dat
x2.3y4.56
x7.7y11.11
x12.5y5.5
```

What we want is to be able to store the x and y coordinates in vectors so that we can plot the points. The lines in this file store combinations of characters and numbers, so the **load** function cannot be used. It is necessary to separate the characters from the numbers so that we can create the vectors. The following is the outline of the script to accomplish this:

fileInpCompare.m

```
fid = fopen('xypoints.dat');
if fid == -1
    disp('File open not successful')
else
   % Create x and y vectors for the data points
   % This part will be filled in using different methods

   % Plot the points
   plot(x,y,'k*')
   xlabel('x')
   ylabel('y')

   % Close the file
   closeresult = fclose(fid);
   if closeresult == 0
       disp('File close successful')
   else
       disp('File close not successful')
   end
end
```

We will now complete the middle part of this script using four different methods: **fgetl**, **fscanf** (two ways), and **textscan**.

To use the **fgetl** function, it is necessary to loop until the end-of-file is reached, reading each line as a character vector, and parsing the character vector into the various components and converting the character vectors containing the actual x and y coordinates to numbers. This would be accomplished as follows:

```
% using fgetl
x = [];
y = [];
aline = fgetl(fid);
while aline ~= -1
     aline = aline(2:end);
     [xstr, rest] = strtok(aline,'y');
     x = [x str2double(xstr)];
     ystr = rest(2:end);
     y = [y str2double(ystr)];
     aline = fgetl(fid);
end
```

Instead, to use the **fscanf** function, we need to specify the format of every line in the file as a character, a number, a character, a number, and the newline character. As the matrix that will be created will store every line from the file in a separate column, the dimensions will be $4 \times n$, where n is the number of lines in the file (and as we do not know that, **inf** is specified instead). The x characters will be in the first row of the matrix (the ASCII equivalent of 'x' in each element), the x coordinates will be in the second row, the ASCII equivalent of 'y' will be in the third row, and the fourth row will store the y coordinates. The code would be:

```
% using fscanf

mat = fscanf(fid, '%c%f%c%f\n', [4, inf]);
x = mat(2,:);
y = mat(4,:);
```

Note that the newline character in the format specifier is necessary. The data file itself was created by typing in the MATLAB Editor/Debugger and, to move down to the next line, the Enter key was used, which is equivalent to the newline character. It is an actual character that is at the end of every line in the file. It is important to note that if the **fscanf** function is looking for a number, it will skip over whitespace characters including blank spaces and newline characters. However, if it is looking for a character, it would read a whitespace character including the newline.

In this case, after reading in 'x2.3y4.56' from the first line of the file, if we had as the format specifier '%c%f%c%f' (without the '\n'), it would then attempt to read again using '%c%f%c%f', but the next character it would read for the first '%c' would be the newline character, and then it would find the 'x' on the second line for the '%f'—not what is intended! (The difference between this and the previous example is that before we read a number followed by a character on each line. Thus, when looking for the next number, it would skip over the newline character.)

Since we know that every line in the file contains the letter 'x' and 'y', not just any random characters, we can build that into the format specifier:

```
% using fscanf method 2

mat = fscanf(fid, 'x%fy%f\n', [2, inf]);
x = mat(1,:);
y = mat(2,:);
```

In this case, the characters 'x' and 'y' are not read into the matrix, so the matrix only has the x coordinates (in the first row) and the y coordinates (in the second row).

Finally, to use the **textscan** function, we could put '%c' in the format specifier for the 'x' and 'y' characters, or build those in as with **fscanf**. If we build

those in, the format specifier essentially specifies that there are four columns in the file, but it will only read the columns with the numbers into column vectors in the cell array *xydat*. The reason that the newline character is not necessary is that, with **textscan**, the format specifier specifies what the columns look like in the file, whereas, with **fscanf**, it specifies the format of every line in the file. Thus, it is a slightly different way of viewing the file format.

```
% using textscan

xydat = textscan(fid,'x%fy%f');
x = xydat{1};
y = xydat{2};
```

To summarize, we have now seen four methods of reading from a file. The function **load** will work only if the values in the file are all the same type and there is the same number of values on every line in the file, so that they can be read into a matrix. If this is not the case, lower-level functions must be used. To use these, the file must be opened first and then closed when the reading has been completed.

The **fscanf** function will read into a matrix, converting the characters to their ASCII equivalents. The **textscan** function will instead read into a cell array that stores each column from the file into separate column vectors of the cell array. Finally, the **fgetl** function can be used in a loop to read each line from the file as a separate character vector; text manipulating functions must then be used to break the character vector into pieces and convert to numbers.

QUICK QUESTION!

If a data file is in the following format, which file input function(s) could be used to read it in?

```
48    25    23    23
12    45     1    31
31    39    42    40
```

Answer: Any of the file input functions could be used, but as the file consists of only numbers and four on each line, the **load** function would be the easiest.

9.4 DATA TRANSFER WITH WEB SITES

MATLAB has functions that allow you to access data that is readily available on many websites. Many companies and government agencies put data on their sites. Examples include weather, census, mapping, transportation schedules, and car sharing sites. There are a lot of concepts related to accessing these data, and there is a lot of jargon in this section. To a degree, this goes beyond the scope of this book, but the possibilities for cool applications are endless and

so it is hoped that this will spark interest in readers, who will then investigate these topics further.

An ***Application Programming Interface (API)*** allows for communication between devices or computers. Device APIs communicate between devices. Remote or web APIs connect computers, for example, a web API may connect a web application to a database. Some APIs are private, and some are public. A ***RESTful*** (Representational State Transfer) API is a particular type of API.

API calls allow one to retrieve data or to edit data in a database. API calls use HTTP requests. HTTP methods include GET, POST, PUT, and DELETE. API calls combine HTTP request messages, or methods, and the desired structure of the response. For example, a GET request allows one to retrieve data from a database based on parameters. In MATLAB, the function **webread** is used for this. The **webwrite** function retrieves data based on an object; this is a POST request. These functions were introduced in R2014b.

Javascript Object Notation, or JSON, is a standard notation that is used to create an object for a POST request, or to standardize the return format of information in a GET request. JSON uses what are called Key Value pairs, which are similar to structure field names and their values in MATLAB.

The **webread** function reads information from a RESTful web service that is specified by a URL. The URL is the API service ***endpoint***. Assuming that the variable *url* stores the URL of a web service as a character vector, the call

```
wdata = webread(url);
```

will read data from the web service, return the content, and store it in a variable *wdata*. The data returned may be in JSON format. ***Query parameters*** may be passed to the **webread** function in the form of Key Value pairs, in order to request specified information, e.g.,

```
wdata = webread(url, Key1, Value1, Key2, Value2);
```

For example, for a stock API, it might be necessary to request information about a particular stock, so for example:

```
wdata = webread(url, 'symbol','acme');
```

This might be done in a loop to get information about multiple stocks; the stock names might be in a cell array that is indexed. If calls to the API are made too quickly, there may be errors, so in some cases it may be useful to put a **pause** statement in the loop so that this does not occur. Also, in some cases it might take a while for the data to be read from the web site. It may be useful to put some form of print statement in the loop to let the user know what is going on, e.g.,

```
Retrieving stock data for 'GOOGL'..
Retrieving stock data for 'AAPL'..
```

Images and audio data can also be read from a RESTful web service; see the documentation on the **webread** function and more on images and sounds in Chapter 13.

The format of the input to **webread** and the format of the output from it are dictated by the web service. The **webread** function itself sometimes translates the returned data from JSON to another format, for example, a struct array.

Options in the form of Key Value pairs can be specified using the **weboptions** function. Without any parameters, **weboptions** creates a weboptions object:

```
>> weboptions
ans =
  weboptions with properties:

      CharacterEncoding: 'auto'
              UserAgent: 'MATLAB 9.4.0.723441 (R2018a)'
                Timeout: 5
               Username: ''
               Password: ''
                KeyName: ''
               KeyValue: ''
            ContentType: 'auto'
          ContentReader: []
              MediaType: 'auto'
          RequestMethod: 'auto'
            ArrayFormat: 'csv'
           HeaderFields: []
    CertificateFilename: 'etc'
```

If a web service expects a Key name and its value, this can be specified. For example, if a web service expects a value for a Key called *timein*, this could be specified as

```
myopt = weboptions('KeyName', 'timein', 'KeyValue', 10);
```

Another option might specify that the format of the output should be a character vector rather than, for example, JSON:

```
myopt = weboptions('ContentType', 'text');
```

The **webwrite** function can also retrieve information from a web service, using a POST request that is based on an object. Frequently, the format of the object is JSON. These requests are more complicated than using **webread**. See the MATLAB documentation for more details and examples.

In R2016b, new functions **jsonencode** and **jsondecode** were introduced that make it easy to convert from MATLAB types to JSON and vice versa. For example, the following takes a struct and encodes it in JSON format and then back to a struct.

```
>> personstruct = struct('Name', 'Harry', 'Age', 33);
>> jsonpers = jsonencode(personstruct)
jsonpers =
     '{"Name":"Harry","Age":33}'
>> newstr = jsondecode(jsonpers)
newstr =
  struct with fields:

    Name: 'Harry'
    Age : 33
```

There are many examples of web services that are available. To find them, begin by looking at the examples in the documentation on the **webread** function and, in general, Web Access from MATLAB.

Some APIs can be accessed easily as seen in this section. Other APIs are public, but require that some form of access key or authentication be presented. In many cases, you must request an access key from a web service, and once it has been returned to you, you must include this key with your request to the web service. Some of these keys are free, but sometimes you must pay for them.

To find APIs, the best method is to use a search browser. For example, if you are interested in getting weather information, you might search for "weather api". Options for different weather web services will appear in the browser. You must read the information in detail about each web service API, including whether or not you need an access key and if so how to get it, the query parameters that are necessary to retrieve data, and the format of the output that will be returned. If you are interested in a particular company, search for the company name, e.g., "company API". There are endless possibilities. There is so much data out there!

■ Explore Other Interesting Features

Reading from and writing to binary files, using the functions **fread**, **fwrite**, **fseek**, and **frewind**. Note that to open a file to both read from it and write to it, the plus sign must be added to the permission (e.g., 'r+').

Use **help load** to find some example MAT-files in MATLAB.

Email can be sent from MATLAB using the **sendmail** function. In order to do this, the **setpref** function must first be used to set preferences for the email address and for the SMTP server information. See the MATLAB documentation on these functions for more details and examples. Similarly, text messages can be sent from MATLAB.

The **dlmread** function reads from an ASCII-delimited file into a matrix; also investigate the **dlmwrite** function.

The Import Tool to import files from a variety of file formats.

In the MATLAB Product Help, enter "Supported File Formats" to find a table of the file formats that are supported, and the functions that read from them and write to them. ■

SUMMARY

COMMON PITFALLS

- Misspelling a file name, which causes a file open to be unsuccessful.
- Using a lower-level file I/O function, when **load** or **save** could be used.
- Forgetting that **fscanf** reads columnwise into a matrix, so every line in the file is read into a column in the resulting matrix.
- Forgetting that **fscanf** converts characters to their ASCII equivalents.
- Forgetting that **textscan** reads into a cell array (so curly braces are necessary to index).
- Forgetting to use the permission 'a' for appending to a file (which means the data already in the file would be lost if 'w' was used!).

PROGRAMMING STYLE GUIDELINES

- Use **load** when the file contains the same kind of data on every line and in the same format on every line.
- Always close files that were opened.
- Always check to make sure that files were opened and closed successfully.
- Make sure that all data are read from a file; e.g., use a conditional loop to loop until the end of the file is reached rather than using a **for** loop.
- Be careful to use the correct formatting specifier when using **fscanf** or **textscan**.
- Store groups of related variables in separate MAT-files.

MATLAB Functions and Commands

xlswrite	fgetl	fscanf	weboptions
xlsread	fgets	textscan	jsonencode
fopen	feof	webread	jsondecode
fclose	fprintf	webwrite	

Exercises

1. Create a spreadsheet that has on each line an integer student identification number followed by three quiz grades for that student. Read that information from the spreadsheet into a matrix and print the average quiz score for each student.
2. The **xlswrite** function can write the contents of a cell array to a spreadsheet. A manufacturer stores information on the weights of some parts in a cell array. Each row stores the part identifier code followed by weights of some sample parts. To simulate this, create the following cell array:

```
>> parts = {'A22',   4.41   4.44   4.39   4.39
            'Z29',   8.88   8.95   8.84   8.92}
```

   Then, write this to a spreadsheet file.
3. A spreadsheet *popdata.xls* stores the population every 20 years for a small town that underwent a boom and then decline. Create this spreadsheet (include the header row) and then read the headers into a cell array and the numbers into a matrix. Plot the data using the header strings on the axis labels.

Year	Population
1920	4021
1940	8053
1960	14994
1980	9942
2000	3385

4. Create a multiplication table and write it to a spreadsheet.
5. Read numbers from any spreadsheet file, and write the variable to a MAT-file.
6. Clear out any variables that you have in your Command Window. Create a matrix variable and two vector variables.
   - Make sure that you have your Current Folder set.
   - Write all variables to a MAT-file
   - Store just the two vector variables in a different MAT-file
   - Verify the contents of your files using **who**.
7. Create a set of random matrix variables with descriptive names (e.g., *ran2by2int*, *ran3by3double*, etc.) for use when testing matrix functions. Store all of these in a MAT-file.
8. What is the difference between a data file and a MAT-file?

9. Write a script "pwcount" that will read in a user's password from a MAT-file, prompt the user for his/her password until it is entered correctly, and then print how many attempts it took until the password was entered correctly. The password is in a variable named "password" in a MAT-file named "pw.mat".
10. Write a script that will prompt the user for the name of a file from which to read. Loop to error-check until the user enters a valid filename that can be opened. (Note: this would be part of a longer program that would actually do something with the file, but for this problem all you have to do is to error-check until the user enters a valid filename that can be read from.)
11. A file "potfilenames.dat" stores potential file names, one per line. The names do not have any extension. Write a script that will print the names of the valid files, once the extension ".dat" has been added. "Valid" means that the file exists in the Current Directory, so it could be opened for reading. The script will also print how many of the file names were valid.
12. A set of data files named "exfile1.dat", "exfile2.dat", etc. has been created by a series of experiments. It is not known exactly how many are there, but the files are numbered sequentially with integers beginning with 1. The files all store combinations of numbers and characters and are not in the same format. Write a script that will count how many lines total are in the files. Note that you do not have to process the data in the files in any way; just count the number of lines.
13. Write a script that will read from a file x and y data points in the following format:

```
x 0 y 1
x 1.3 y 2.2
```

    The format of every line in the file is the letter 'x', a space, the x value, space, the letter 'y', space, and the y value. First, create the data file with 10 lines in this format. Do this by using the Editor/Debugger, then File Save As *xypts.dat*. The script will attempt to open the data file and error-check to make sure it was opened. If so, it uses a **for** loop and **fgetl** to read each line as a character vector. In the loop, it creates x and y vectors for the data points. After the loop, it plots these points and attempts to close the file. The script should print whether or not the file was successfully closed.
14. Modify the script from the previous problem. Assume that the data file is in exactly that format, but do not assume that the number of lines in the file is known. Instead of using a **for** loop, loop until the end of the file is reached. The number of points, however, should be in the plot title.
15. Write a script "custformat" that will read names from a file "customers.txt" in the form "Last, First" (one space in between) and will print them in the form "First Last". For example, IF the file stores the following:

```
>> type customers.txt
Jones, Juanita
Kowalcyk, Kimberly
Ant, Adam
Belfry, Betty
```

Executing the script would produce the following:

```
>> custformat
Juanita Jones
Kimberly Kowalcyk
Adam Ant
Betty Belfry
```

16. Create a data file to store blood donor information for a biomedical research company. For every donor, store the person's name, blood type, Rh factor, and blood pressure information. The Blood type is either A, B, AB, or O. The Rh factor is + or −. The blood pressure consists of two readings: systolic and diastolic (both are **double** numbers). Write a script to read from your file into a data structure and print the information from the file.
17. Assume that a file named *testread.dat* stores the following:

```
110x0.123y5.67z8.45
120x0.543y6.77z11.56
```

Assume that the following are typed SEQUENTIALLY. What would the values be?

```
tstid = fopen('testread.dat')
fileline = fgetl(tstid)
[beg, endline] = strtok(fileline,'y')
length(beg)
feof(tstid)
```

18. Create a data file to store information on hurricanes. Each line in the file should have the name of the hurricane, its speed in miles per hour, and the diameter of its eye in miles. Then, write a script to read this information from the file and create a vector of structures to store it. Print the name and area of the eye for each hurricane.
19. Create a file "parts_inv.dat" that stores on each line a part number, cost, and quantity in inventory, in the following format:

```
123 5.99 52
```

Use **fscanf** to read this information and print the total dollar amount of inventory (the sum of the cost multiplied by the quantity for each part).

20. Students from a class took an exam for which there were 2 versions, marked either A or B on the front cover (½ of the students had version A, ½ Version B). The exam results are stored in a file called "exams.dat", which has on

each line the version of the exam (the letter 'A' or 'B') followed by a space followed by the integer exam grade. Write a script that will read this information from the file using **fscanf**, and separate the exam scores into two separate vectors: one for Version A, and one for Version B. Then, the grades from the vectors will be printed in the following format (using **disp**).

```
A exam grades:
  99    80     76

B exam grades:
  85    82    100
```

Note: no loops or selection statements are necessary!

21. Create a file which stores on each line a letter, a space, and a real number. For example, it might look like this:

```
e 5.4
f 3.3
c 2.2
```

Write a script that uses **textscan** to read from this file. It will print the sum of the numbers in the file. The script should error-check the file open and close, and print error messages as necessary.

22. Write a script to read in division codes and sales for a company from a file that has the following format:

```
A    4.2
B    3.9
```

Print the division with the highest sales.

23. A data file is created as a **char** matrix and then saved to a file; for example,

```
>> cmat = char('hello', 'ciao', 'goodbye')
cmat =
hello
ciao
goodbye
>> save stringsfile.dat cmat -ascii
```

Can the **load** function be used to read this in? What about **textscan**?

24. Create a file of text as in the previous exercise, but create the file by opening a new file, type in the text, and then save it as a data file. Can the **load** function be used to read this in? What about **textscan**?

25. Write a script that creates a cell array of character vectors, each of which is a two-word phrase. The script is to write the first word of each phrase to a file "examstrings.dat". You do not have to error-check on the file open or file close. The script should be general and should work for any cell array containing two-word phrases.

26. The Wind Chill Factor (WCF) measures how cold it feels with a given air temperature (T, in degrees Fahrenheit) and wind speed (V, in miles per hour). One formula for the WCF follows:

$$\text{WCF} = 35.7 + 0.6\,T - 35.7\,(V^{0.16}) + 0.43\,T\,(V^{0.16})$$

Create a table showing WCFs for temperatures ranging from −20 to 55 in steps of 5, and wind speeds ranging from 0 to 55 in steps of 5. Write this to a file *wcftable.dat*. If you have version R2016a or later, write the script as a live script.

27. Write a script that will loop to prompt the user for *n* circle radii. The script will call a function to calculate the area of each circle and will write the results in sentence form to a file.

28. Write a function "write_exclaims" that will write exclamation points in the specified format to a file "exclamations.txt". The function will receive two integer input arguments, the number of rows to write, and the number of exclamation points to write on each row. It may be assumed that the input arguments are both positive integers. Here is an example of calling the function and the resulting file.

```
>> r = 4;
>> ex = 10;
>> write_exclaims(r,ex)
>> type exclamations.txt
1: !!!!!!!!!!
2: !!!!!!!!!!
3: !!!!!!!!!!
4: !!!!!!!!!!
>>
```

29. Create a file that has some college department names and enrollments. For example, it might look like this:

```
Aerospace 201
Mechanical 66
```

Write a script that will read the information from this file and create a new file that has just the first four characters from the department names, followed by the enrollments. The new file will be in this form:

```
Aero 201
Mech 66
```

30. For a project, some biomedical engineering students are monitoring the target heart rate for subjects. A simple calculation of the target heart rate (THR) for a moderately active person is

$$\text{THR} = (220 - A)*.6$$

where A is the person's age. A file "patient_ages.txt" stores, on each line, a person's name and age in the following format:

```
>> type patient_ages.txt
Bob 22
Mary 18
Joan 39
>>
```

Write a script that will read the information from the "patient_ages.txt" file, and for each person, write their THR to another file "targets.txt". For example, IF the "patient_ages.txt" file is as shown above, as a result of executing the script, the "targets.txt" file would store

```
>> type targets.txt
Bob's thr is 118.8
Mary's thr is 121.2
Joan's thr is 108.6
```

31. Environmental engineers are trying to determine whether the underground aquifers in a region are being drained by a new spring water company in the area. Well depth data has been collected every year at several locations in the area. Create a data file that stores on each line the year, an alphanumeric code representing the location, and the measured well depth that year. Write a script that will read the data from the file and determine whether or not the average well depth has been lowered.
32. A program is being written to store and manipulate information on some people, including their name, height (in feet and inches, both integers), and weight in pounds. This information is stored in a file named "fxxpeople.dat". Each line in the file stores the information for one person; there is one blank space between each piece of information in the file. For example, the file might look like this:

```
Sammie 5 10 182
Richard 6 5 222.5
```

A script has been written to read this information in, store it in a vector of structures, and then print it out. For now, error-checking on the file open and close has been omitted for simplicity. Write the "parseline" function that will take each line from the file and return a structure in the required format. For example, IF the file is as shown above, the result from executing the script would be:

```
>> fxx
Sammie is 5' 10" tall and weighs 182.0 pounds
Richard is 6' 5" tall and weighs 222.5 pounds
```

fxx.m

```
fid = fopen('fxxpeople.dat');
i = 0;
aline = fgetl(fid);
while aline ~= -1
     i = i + 1;
     astruct = parseline(aline);
     people(i) = astruct;
     aline = fgetl(fid);
end

for i = 1:length(people)
     fprintf('%s is %d'' %d" tall and weighs %.1f pounds\n',...
     people(i).name, people(i).height.feet, people(i).height.inches,...
     people(i).weight)
end
fclose(fid);
```

33. Write a menu-driven program that will read in an employee data base for a company from a file and do specified operations on the data. The file stores the following information for each employee:

    - Name
    - Department
    - Birth Date
    - Date Hired
    - Annual Salary
    - Office Phone Extension

    You are to decide exactly how this information is to be stored in the file. Design the layout of the file, and then create a sample data file in this format to use when testing your program. The format of the file is up to you. However, space is critical. Do not use any more characters in your file than you have to! Your program is to read the information from the file into a data structure, and then display a menu of options for operations to be done on the data. You may not assume in your program that you know the length of the data file. The menu options are:

    1. Print all of the information in an easy-to-read format to a new file.
    2. Print the information for a particular department.
    3. Calculate the total payroll for the company (the sum of the salaries).
    4. Find out how many employees have been with the company for N years (N might be 10, for example).
    5. Exit the program.

34. Practice with JSON format. Create a nested struct and encode it into JSON format.
35. Read the documentation page on **webread**. There are examples of API calls; try one of them and examine the structure of the result.
36. Use a browser to find a free API of interest to you. Read the documentation page carefully before attempting to retrieve data from it.