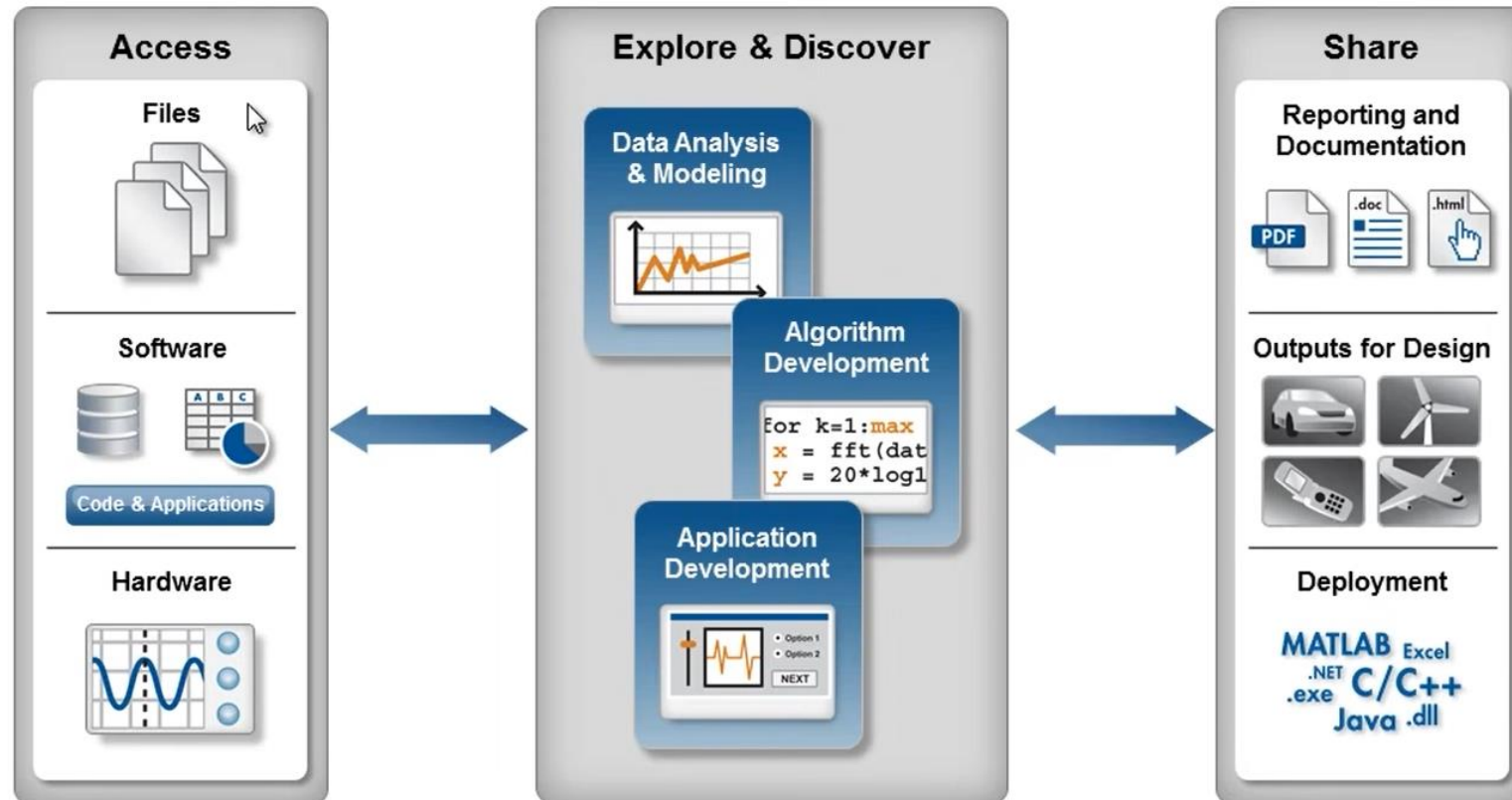


# Introduction to MATLAB Programming

File input and output

# Introduction

## Data Analysis Tasks



# File input/output

There are basically three different operations, or *modes*, on files. Files can be:

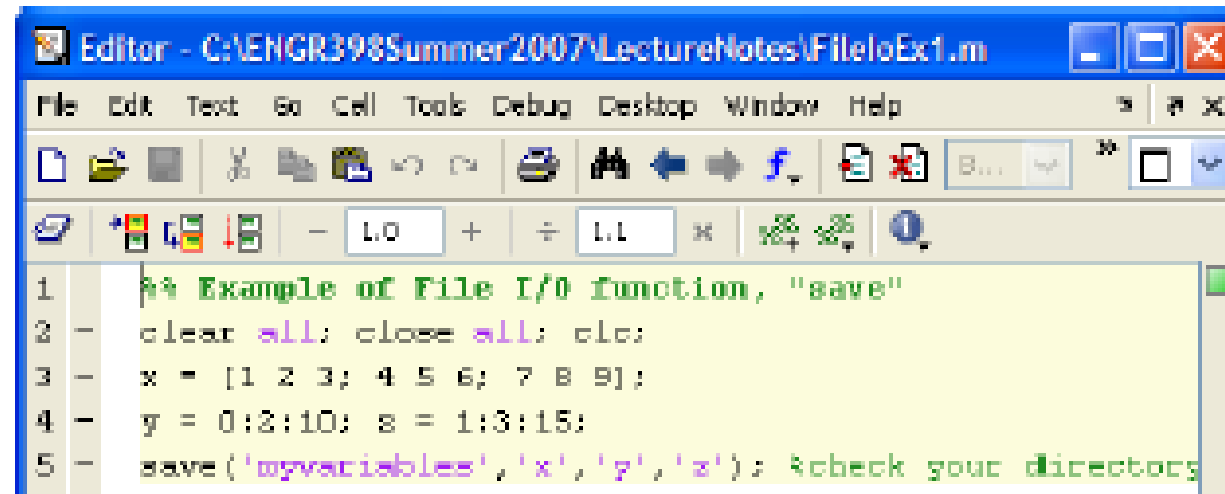
- Read from
- Written to
- *Appended* to

Writing to a file means writing to a file, from the beginning. Appending to a file is also writing, but starting at the end of the file rather than the beginning. In other words, appending to a file means adding to what was already there.

## Save and Load \*.mat files

- `save filename var1 var2 var3`
- `load filename`
- The save command saves data from the workspace into an `*.mat` file.
- The load command loads the variables stored in the `filename.mat` file previously saved.
- `filename` is the name of the file (a string) and `var1`, `var2`, `var3` are the variables to be saved in a MATLAB readable file.
- If the variables are not listed after the filename, save saves all the variables in the workspace.

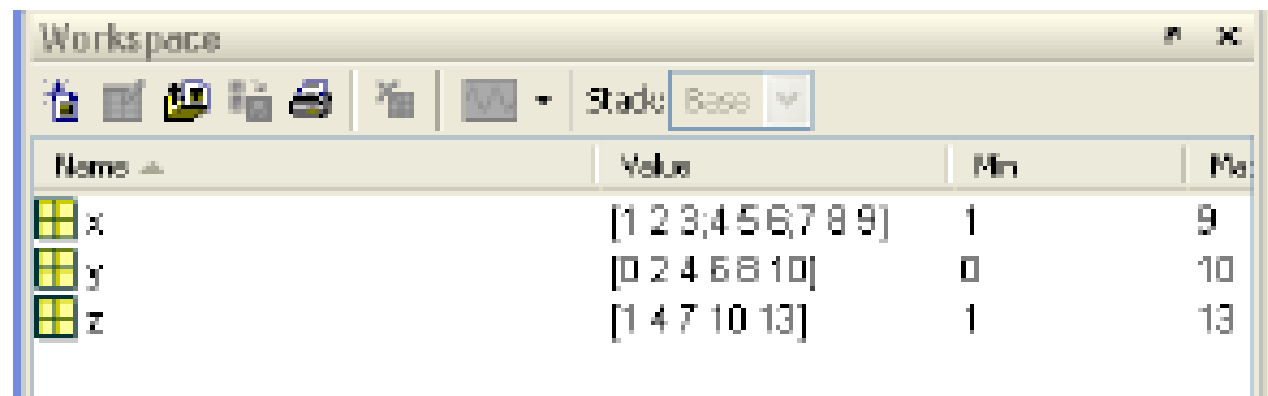
# Saving to MATLAB .mat File



The image shows a MATLAB Editor window titled "Editor - C:\ENGR398Summer2007\LectureNotes\FileIoEx1.m". The menu bar includes File, Edit, Text, Go, Cell, Tools, Debug, Desktop, Window, and Help. The toolbar contains icons for file operations and editing. The script content is as follows:

```
1 %% Example of File I/O function, "save"
2 - clear all; close all; clc;
3 - x = [1 2 3; 4 5 6; 7 8 9];
4 - y = 0:2:10; z = 1:3:15;
5 - save('myvariables','x','y','z'); %check your directory
```


After executing the .m script



The image shows the MATLAB Workspace window. It has a toolbar with icons for workspace operations and a "State" dropdown menu set to "Base". The workspace contains three variables: x, y, and z. The table below represents the data shown in the workspace:

Name	Value	Min	Max
x	[1 2 3; 4 5 6; 7 8 9]	1	9
y	[0 2 4 6 8 10]	0	10
z	[1 4 7 10 13]	1	13

## Loading From a .mat File



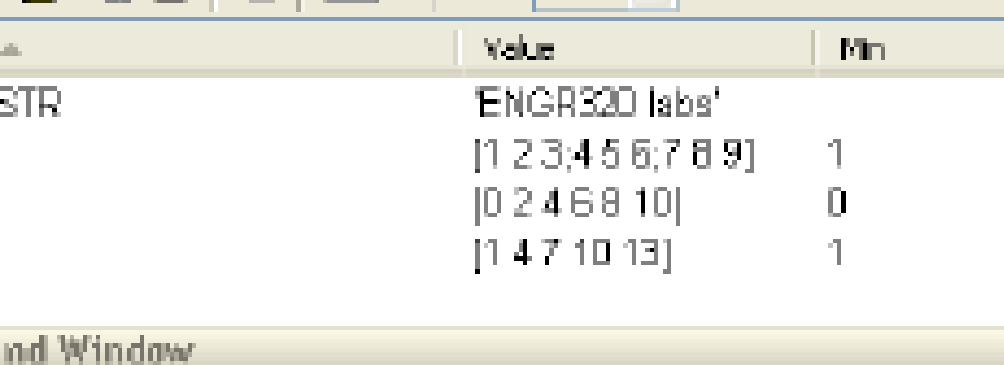
The screenshot shows the MATLAB Editor interface. The title bar reads "Editor - C:\ENGR398Summer2007\LectureNotes\...". The menu bar includes File, Edit, Test, Go, Cell, Tools, Debug, Desktop, and Window. The toolbar contains icons for file operations (New, Open, Save, Print, etc.) and editing (Undo, Redo, Cut, Paste, etc.). The main workspace area displays a script with the following code:

```

1 %% File I/O function, 'load'
2 - clear all; close all;
3 - MySTR = 'ENGR320 labs';
4 - whos
5 - load('myvariables.mat');
6 - whos %Check what variables are loa

```

After executing the “m-file”



The screenshot shows the MATLAB environment. The **Workspace** window displays the following variables:

Name	Value	Min	Max
MySTR	'ENGR320 labs'		
x	[1 2 3 4 5 6; 7 8 9]	1	9
y	[0 2 4 6 8 10]	0	10
z	[1 4 7 10 13]	1	13

The **Command Window** shows the execution of the `FileIoEx2` script, followed by a display of variable properties:

```
>> FileIoEx2
```

Name	Size	Bytes	Class	Attributes
MySTR	1x12	24	char	

Name	Size	Bytes	Class	Attributes
MySTR	1x12	24	char	
x	3x3	72	double	
y	1x6	48	double	
z	1x5	40	double	

# 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* already has been stored in the file 'sess2.mat' as shown earlier, 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 Command Window 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 Command Window.

# 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 been created yet, 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
```



# Writing data to a file

The `save` function can be used to write data from a matrix to a data file, or to append to a data file. The format is:

```
save filename matrixvariablename -ascii
```

The `-ascii` qualifier is used when creating a text or data file. The following creates a matrix and then saves the values of the matrix variable to a data file called `testfile.dat`:

```
>> mymat = rand(2,3)
mymat =
    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919
>> save testfile.dat mymat -ascii
```

This creates a file called `testfile.dat` that stores the numbers

```
0.4565  0.8214  0.6154
0.0185  0.4447  0.7919
```

The `type` command can be used to display the contents of the file; notice that scientific notation is used:

```
>> type testfile.dat
4.5646767e-001  8.2140716e-001  6.1543235e-001
1.8503643e-002  4.4470336e-001  7.9193704e-001
```

Note: If the file already exists, the `save` function will overwrite it; `save` always begins writing from the beginning of a file.

# Appending data to a data file

Once a text file exists, data can be appended to it. The format is the same as previously, with the addition of the qualifier `-append`. For example, the following creates a new random matrix and appends it to the file just created:

```
>> mymat = rand(3,3)
mymat =
    0.9218    0.4057    0.4103
    0.7382    0.9355    0.8936
    0.1763    0.9169    0.0579

>> save testfile.dat mymat -ascii -append
```

This results in the file `testfile.dat` containing

```
0.4565    0.8214    0.6154
0.0185    0.4447    0.7919
0.9218    0.4057    0.4103
0.7382    0.9355    0.8936
0.1763    0.9169    0.0579
```

**Note:** Although technically any size matrix could be appended to this data file, in order to be able to read it back into a matrix later there would have to be the same number of values on every row.

# Reading from a file

Once a file has been created (as previously), it can be read into a matrix variable. If the file is a data file, the load function will read from the file filename.ext (e.g., the extension might be .dat) and create a matrix with the same name as the file. For example, if the data file testfile.dat had been created as shown in the previous section, this would read from it:

```
>> clear
>> load testfile.dat
>> who
Your variables are:
testfile
>> testfile
testfile =

    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919
    0.9218    0.4057    0.4103
    0.7382    0.9355    0.8936
    0.1763    0.9169    0.0579
```

Note: The load command works only if there are the same number of values in each line, so that the data can be stored in a matrix, and the save command only writes from a matrix to a file. If this is not the case, lower-level file I/O functions must be used;

Prompt the user for the number of rows and columns of a matrix, create a matrix with that many rows and columns of random numbers, and write it to a file.

# Exercise

As an example, a file called `timetemp.dat` stores two lines of data. The first line is the times of day, and the second line is the recorded temperature at each of those times. The first value of 0 for the time represents midnight. For example, the contents of the file might be:

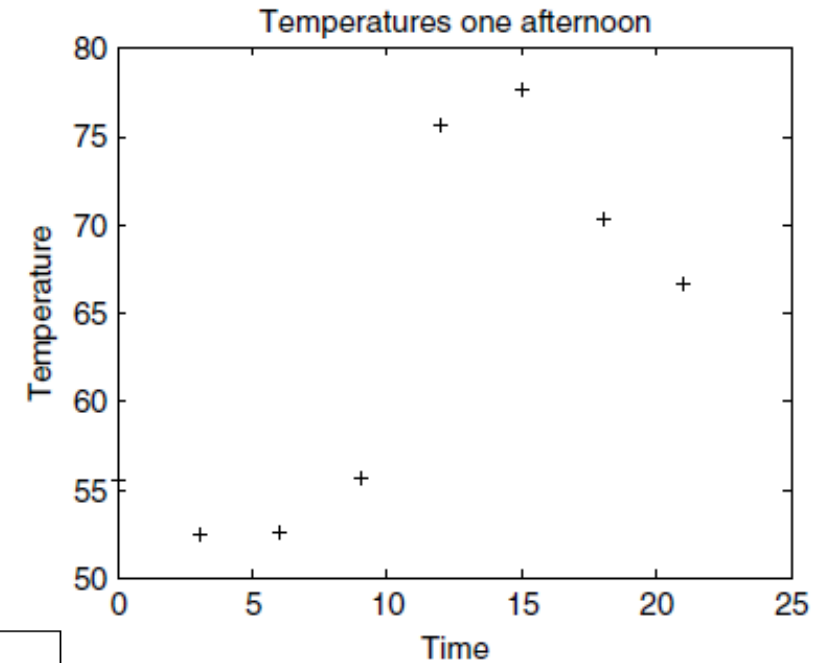
0	3	6	9	12	15	18	21
55.5	52.4	52.6	55.7	75.6	77.7	70.3	66.6

The following script loads the data from the file into a matrix called `timetemp`. It then separates the matrix into vectors for the time and temperature, and then plots the data using black + symbols.

```
% This reads time and temperature data for an afternoon
% from a file and plots the data
load timetemp.dat

% The times are in the first row, temps in the second row
time = timetemp(1,:);
temp = timetemp(2,:);

% Plot the data and label the plot
plot(time,temp,'k+')
xlabel('Time')
ylabel('Temperature')
title('Temperatures one afternoon')
```



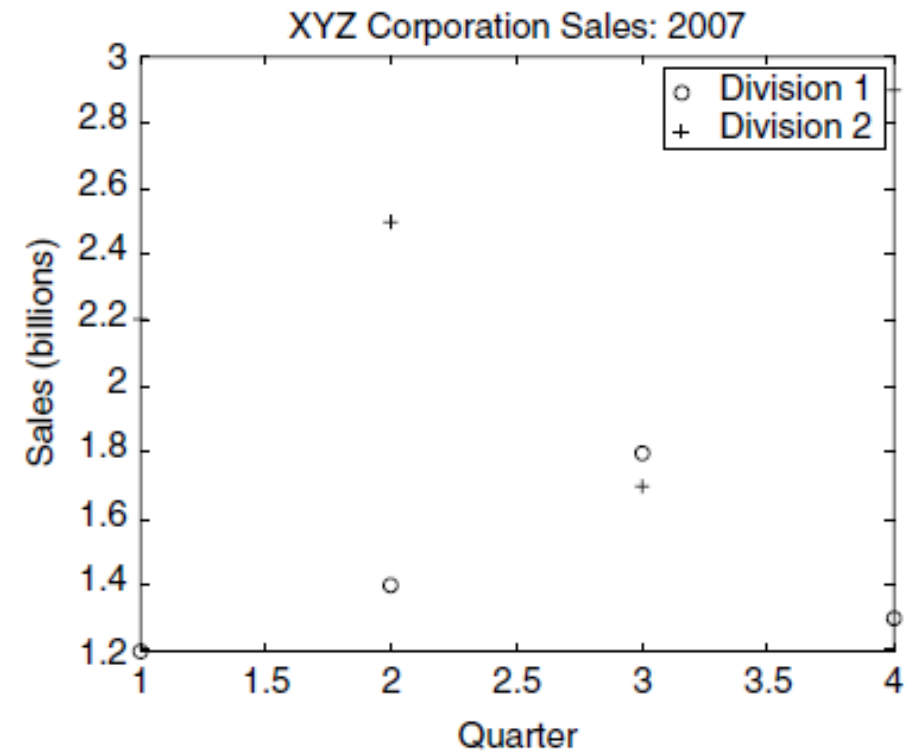
**FIGURE 2.5**  
*Plot of temperature data  
from a file.*

# Practice

The sales (in billions) for two separate divisions of the XYZ Corporation for each of the four quarters of 2007 are stored in a file called salesfigs.dat:

```
1.2 1.4 1.8 1.3
2.2 2.5 1.7 2.9
```

1. Create this file (just type the numbers in the editor, and Save As salesfigs.dat).
2. Load the data from the file into a matrix.
3. Write a script that will
  - separate this matrix into two vectors
  - create the plot seen in Figure 2.6 (which uses o's and +'s as the plot symbols):



**FIGURE 2.6**

*Plot of sales data from file.*

# Practice

Sometimes files are not in the format that is desired. For example, a file `expresults.dat` has been created that has some experimental results, but the order of the values is reversed in the file:

7	55.2
6	51.9
5	49.5
4	53.4
3	44.3
2	50.0
1	55.5

How could we create a new file that reverses the order?

```
>> load expresults.dat
```

```
>> expresults
```

```
expresults =
```

7.0000	55.2000
6.0000	51.9000
5.0000	49.5000
4.0000	53.4000
3.0000	44.3000
2.0000	50.0000
1.0000	55.5000

```
>> correctorder = flipud(expresults)
```

```
correctorder =
```

1.0000	55.5000
2.0000	50.0000
3.0000	44.3000
4.0000	53.4000
5.0000	49.5000
6.0000	51.9000
7.0000	55.2000

```
>> save neworder.dat correctorder - ascii
```

# Lower level file I/O functions

When reading from a data file, the **load** function works as long as the data in the file is “regular”—in other words, the same kind of data on every line and in the same format on every line—so that it 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. When the reading has been completed, the file must be closed.

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

The steps involved are:

- Open the file.
- Read from the file, write to the file, or append to the file.
- Close the file.



# Opening and Closing a file (1)

Files are opened with the **fopen** function. By default, the **fopen** function opens a file for reading. If another mode is desired, a permission string is used to specify which mode (e.g., writing or appending). The **fopen** function returns  $-1$  if it is not successful in opening the file, or an integer value, which 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 string');
```

The permission strings include:

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

See **help fopen** for others.



# Opening and Closing a file (2)

After the **fopen** is attempted, the value returned should be tested to make sure that the file was successfully opened. For example, if the file does not exist, the **fopen** will not be successful. Since 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
```

Files should be closed when the program has finished reading from or writing 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' to the **fclose** function. The general forms are:

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

This should also be checked with an **if-else** statement to make sure it was successful.

# Reading from files (1)

There are several lower level functions that read from files. 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 it in a cell array. The **fgetl** and **fgets** functions both read strings 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. All these functions require first opening the file, and then closing it when finished.

Since the **fgetl** and **fgets** functions read one line at a time, these functions are typically in some form of a loop. 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; they will read in the entire file automatically but into a data structure.

# Reading from files (2)

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

- Attempt to open the file; check to make sure the file open was successful.
- If opened, loop until the end of the file is reached. For each line in the file,
  - read it into a string
  - manipulate the data
- Attempt to close the file; check to make sure the file close was successful.

# Reading from files (3)

The generic code to accomplish this is:

```
fid = fopen('filename');
if fid == -1
    disp('File open not successful')
else
    while feof(fid) == 0
        % Read one line into a string variable
        aline = fgetl(fid);
        % Use string functions to extract numbers, strings,
        % etc. from the line
        % Do something with the data!
    end
    closeresult = fclose(fid);
    if closeresult == 0
        disp('File close successful')
    else
        disp('File close not successful')
    end
end
```

# Reading from files (4)

The permission string could be included in the call to the **fopen** function, for example,

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

but is not necessary since reading is the default. The condition on the **while** loop can be interpreted as saying “while the file end-of-file is false.” Another way to write this is

```
while ~feof(fid)
```

which is interpreted as “while we’re not at the end of the file.”

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

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

# Reading from files (5)

The **load** function would not be able to read this into a matrix since it contains both numbers and text. Instead, the **fgetl** function can be used to read each line as a string and then string functions are used to separate the numbers and characters. For example, the following just reads each line and prints the number with 2 decimal places and then the rest of the string:

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
    while feof(fid) == 0
        aline = fgetl(fid);
        % Separate each line into the number and character
```

Here 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
```

```
        % code and convert to a number before printing
        [num charcode] = strtok(aline);
        fprintf('%.2f %s\n', str2num(num), charcode)
    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
```

# Practice

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

# Reading from files (6)

Instead of using the `fgetl` function to read one line, once a file has been opened the `fscanf` function could be used to read in 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])
```

```
>> 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);
```

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



# Reading from files (7)

The format string `'%f %c'` specifies that on each line there is a **double** value followed by a space followed by a character. This creates a  $1 \times 2$  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 =
    [5x1 double] [5x1 char]
>> subjdata{1}
ans =
    5.3000
    2.2000
    3.3000
    4.4000
    1.1000
```

```
>> subjdata{2}
ans =
    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
```

# Reading from files (8)

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
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

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

# Writing to files (1)

There are several lower level functions that can write to files. 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. 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 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 you do not want to see that number, suppress the output with a

# Writing to files (2)

semicolon as shown here. (**Note:** When writing to the screen, the value returned by `fprintf` is not seen, but could be stored in a variable.)

Here is an example of writing to a file named 'tryit.txt':

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

The permission string in the call to the **fopen** function specifies that the file is opened for writing to it. Just as 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.

# Writing to files (3)

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('tryt.txt');
>> while ~feof(fid)
    aline = fgetl(fid)
end
aline =
The loop variable is 1
aline =
The loop variable is 2
aline =
The loop variable is 3
>> fclose(fid);
```

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

```
>> mat = randint(2,4,[5 20])
mat =
    20    14    19    12
     8    12    17     5
>> fid = fopen('randmat.dat','w');
>> fprintf(fid,'%d %d\n',mat);
>> fclose(fid);
```

# Writing to files (4)

Since 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 you want this 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

Create a  $3 \times 5$  matrix of random integers, each in the range from 1 to 100. Write this to a file called 'myrandmat.dat' in a  $3 \times 5$  format using **fprintf**, so that the file appears identical to the original matrix. Load the file to confirm that it was created correctly.



# Appending to files

The `fprintf` function can also be used to append to an existing file. The permission string is 'a', for example,

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

Unlike text files, data doesn't have to be in the same format as what is already in the file when appending.

# Writing and reading spreadsheet files (1)

The MATLAB functions `xlswrite` and `xlsread` will write to and read from spreadsheet files that have the extension `.xls`. For example, the following will create a  $5 \times 3$  matrix of random integers, and then write it to a spreadsheet file called 'ranexcel.xls' that has five rows and three columns:

```
>> ranmat = randint(5,3,[1 100])
```

```
ranmat =
```

```
    96    77    62
```

```
    24    46    80
```

```
    61     2    93
```

```
    49    83    74
```

```
    90    45    18
```

```
>> xlswrite('ranexcel',ranmat)
```

# Writing and reading spreadsheet files (2)

The `xlsread` function will read from a spreadsheet file. For example, to read from the file just created:

```
>> 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 filename is the default, so it can be omitted.

These are shown in their most basic forms, when the matrix 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

# Writing and reading spreadsheet files (3)

```
>> [nums, txt] = xlsread('texttest.xls')
nums =
    123
    333
    432
    987
txt =
    'a'  ''  'Cindy'
    'b'  ''  'Suzanne'
    'c'  ''  'David'
    'd'  ''  '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  $4 \times 3$ . It has three columns since the file had three columns, but since 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 strings.

```
>> txt{1,2}
ans =
    ''

>> txt{1,3}
ans =
Cindy
```

# Writing and reading spreadsheet files (4)

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
```

# Import spreadsheet data using readtable (1)

```
readtable('datafile for test.xlsx')
```

You can also select the range of data to import by specifying the range parameter. For example, read the first five rows and three columns of the spreadsheet.

```
A=readtable('datafile for test.xlsx','Range','A1:C5')
```

# Import spreadsheet data using readtable (2)

In addition to tables, you can import your spreadsheet data into the MATLAB workspace as a timetable, a numeric matrix, a cell array, or separate column vectors. Based on the data type you need, use one of these functions.

Data Type of Output	Function
Timetable	readtimetable
Numeric Matrix	readmatrix
Cell Array	readcell
Separate Column Vectors	readvars

## Read Spreadsheet Data into Matrix

Import numeric data from `basic_matrix.xls` into a matrix.

```
M = readmatrix('basic_matrix.xls')
```

M = 5×4

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

You can also select the data to import from the spreadsheet by specifying the **Sheet** and **Range** parameters. For example, specify the **Sheet** parameter as `'Sheet1'` and the **Range** parameter as `'B1:D3'`. The `readmatrix` function reads a 3-by-3 subset of the data, starting at the element in the first row and second column of the sheet named `'Sheet1'`.

```
M = readmatrix('basic_matrix.xls','Sheet','Sheet1','Range','B1:D3')
```

M = 3×3

```
8     3     1
4     7     3
6     7    10
```

# Write Data to Excel Spreadsheets

## Write Tabular Data to Spreadsheet File

For example, create a sample table of column-oriented data and display the first five rows.

```
load patients.mat
T = table(LastName, Age, Weight, Smoker);
T(1:5, :)
```

```
ans=5x4 table
    LastName    Age    Weight    Smoker
    _____    ____    _____    _____
    {'Smith'   }    38        176        true
    {'Johnson' }    43        163        false
    {'Williams'}    38        131        false
    {'Jones'   }    40        133        false
    {'Brown'   }    49        119        false
```

Write table `T` to the first sheet in a new spreadsheet file named `patientdata.xlsx`,

```
filename = 'patientdata.xlsx';
writetable(T, filename, 'Sheet', 1, 'Range', 'D1')
```

Write the table `T` without the variable names to a new sheet called `'MyNewSheet'`. To write the data without the variable names, specify the name-value pair `WriteVariableNames` as `false`.

```
writetable(T, filename, 'Sheet', 'MyNewSheet', 'WriteVariableNames', false);
```



# Write Numeric and Text Data to Spreadsheet File

To export a numeric array and a cell array to a Microsoft Excel spreadsheet file, use the `writematrix` or `writecell` functions. You can export data in individual numeric and text workspace variables to any worksheet in the file, and to any location within that worksheet. By default, the import functions write your matrix data to the first worksheet in the file, starting at cell **A1**.

For example, create a sample array of numeric data, **A**, and a sample cell array of text and numeric data, **C**.

```
A = magic(5)
C = {'Time', 'Temp'; 12 98; 13 'x'; 14 97}
```

A =

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

C =

'Time'	'Temp'
[ 12]	[ 98]
[ 13]	'x'
[ 14]	[ 97]

Write array **A** to the 5-by-5 rectangular region, **E1:I5**, on the first sheet in a new spreadsheet file named **testdata.xlsx**.

```
filename = 'testdata.xlsx';
writematrix(A,filename,'Sheet',1,'Range','E1:I5')
```

Write cell array **C** to a rectangular region that starts at cell **B2** on a worksheet named **Temperatures**. You can specify range using only the first cell.

```
writecell(C,filename,'Sheet','Temperatures','Range','B2');
```


# Import Text Files

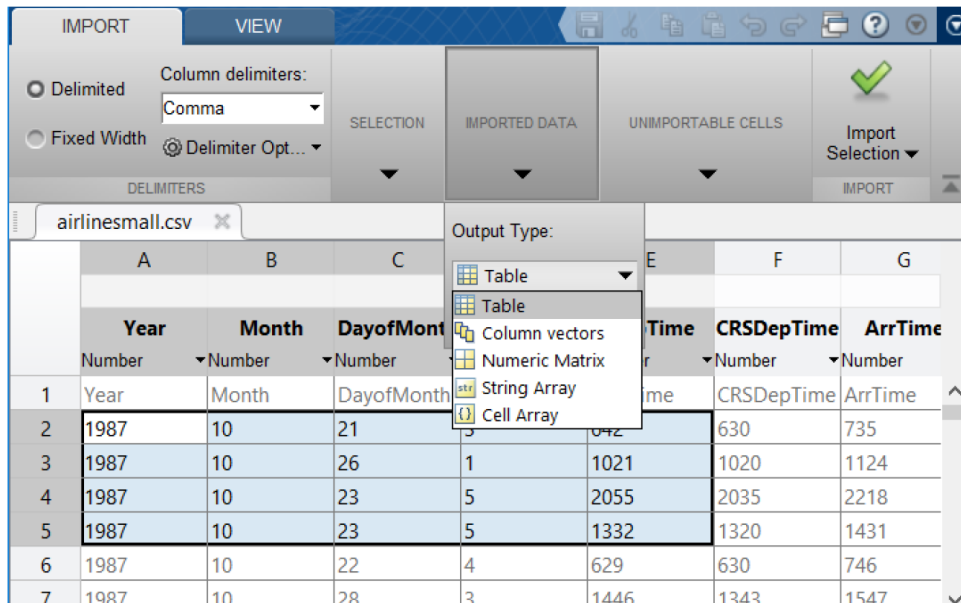
Text files often contain a mix of numeric and text data as well as variable and row names, which is best represented in MATLAB as a table. You can import tabular data from text files into a table using the **Import Tool** or the `readtable` function.

## Import Text Files Using the Import Tool

### Import Text Files Using the Import Tool

The **Import Tool** allows you to import into a table or other data type. For example, read a subset of data from the sample file `airlinesmall.csv`. Open the file using the **Import Tool** and select options such as the range of data to import and the output type. Then, click on the **Import Selection**

button  to import the data into the MATLAB workspace.



## Import Text Files Using readtable

Alternatively, you can read tabular data from a text file into a table using the `readtable` function with the file name, for example:

```
T = readtable('airlinesmall.csv');
```

Display the first five rows and columns from the table.

```
T(1:5,1:5)
```

```
ans =
```

5x5 table

Year	Month	DayofMonth	DayOfWeek	DepTime
1987	10	21	3	{ '642' }
1987	10	26	1	{ '1021' }
1987	10	23	5	{ '2055' }
1987	10	23	5	{ '1332' }
1987	10	22	4	{ '629' }

# Import Numeric Data from Text Files into Matrix

Import numeric data as MATLAB arrays from files stored as comma-separated or delimited text files.

## Import Comma-Separated Data

This example shows how to import comma-separated numeric data from a text file. Create a sample file, read all the data in the file, and then read only a subset starting from a specified location.

Create a sample file named `ph.dat` that contains comma-separated data and display the contents of the file.

```
rng('default')
A = 0.9*randi(99,[3 4]);
writematrix(A,'ph.dat','Delimiter','(',')
type('ph.dat')
```

```
72.9,81.9,25.2,86.4
81,56.7,49.5,14.4
11.7,9,85.5,87.3
```

Read the file using the `readmatrix` function. The function returns a 3-by-4 `double` array containing the data from the file.

```
M = readmatrix('ph.dat')
```

```
M = 3x4
```

```
    72.9000    81.9000    25.2000    86.4000
    81.0000    56.7000    49.5000    14.4000
    11.7000     9.0000    85.5000    87.3000
```

## Import Delimited Numeric Data

This example shows how to import numeric data delimited by any single character using the `writematrix` function. Create a sample file, read the entire file, and then read a subset of the file starting at the specified location.

Create a tab-delimited file named `num.txt` that contains a 4-by-4 numeric array and display the contents of the file.

```
rng('default')  
A = randi(99,[4,4]);  
writematrix(A,'num.txt','Delimiter','\t')  
type('num.txt')
```

```
81    63    95    95  
90    10    96    49  
13    28    16    80  
91    55    97    15
```

Read the entire file. The `readmatrix` function determines the delimiter automatically and returns a 4-by-4 `double` array.

# Write Data to Text Files

## Export Table to Text File

You can export tabular data from MATLAB® workspace into a text file using the `writetable` function. Create a sample table, write the table to text file, and then write the table to text file with additional options.

Create a sample table, `T`, containing the variables `Pitch`, `Shape`, `Price` and `Stock`.

```
Pitch = [0.7;0.8;1;1.25;1.5];  
Shape = {'Pan';'Round';'Button';'Pan';'Round'};  
Price = [10.0;13.59;10.50;12.00;16.69];  
Stock = [376;502;465;1091;562];  
T = table(Pitch,Shape,Price,Stock)
```

```
T=5x4 table  
    Pitch    Shape    Price    Stock  
    _____    _____    _____    _____  
    0.7    {'Pan' }    10    376  
    0.8    {'Round' }    13.59    502  
    1    {'Button' }    10.5    465  
    1.25    {'Pan' }    12    1091  
    1.5    {'Round' }    16.69    562
```

Export the table, `T`, to a text file named `tabledata.txt`. View the contents of the file. By default, `writetable` writes comma-separated data, includes table variable names as column headings.

```
writetable(T,'tabledata.txt');  
type tabledata.txt
```

```
Pitch,Shape,Price,Stock  
0.7,Pan,10,376  
0.8,Round,13.59,502  
1,Button,10.5,465  
1.25,Pan,12,1091  
1.5,Round,16.69,562
```

# Export Numeric Array to Text File

You can export a numerical array to a text file using `writematrix`.

Create a numeric array A.

```
A = magic(5)/10
```

```
A = 5x5
```

1.7000	2.4000	0.1000	0.8000	1.5000
2.3000	0.5000	0.7000	1.4000	1.6000
0.4000	0.6000	1.3000	2.0000	2.2000
1.0000	1.2000	1.9000	2.1000	0.3000
1.1000	1.8000	2.5000	0.2000	0.9000

Write the numeric array to `myData.dat` and specify the delimiter to be `' ; '`. Then, view the contents of the file.

```
writematrix(A, 'myData.dat', 'Delimiter', ';')  
type myData.dat
```

```
1.7;2.4;0.1;0.8;1.5  
2.3;0.5;0.7;1.4;1.6  
0.4;0.6;1.3;2;2.2  
1;1.2;1.9;2.1;0.3  
1.1;1.8;2.5;0.2;0.9
```

# MATLAB Functions and Commands

MATLAB Functions and Commands			
fopen fclose fscanf	fgetl fgets feof	textscan fprintf xlswrite	xlsread



# Summary

## Common Pitfalls

- Misspelling a filename, 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 string 'a' for appending to a file (which means the data already in the file would be lost!)

# Summary

## 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 is read from a file; for example, 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 string when using **fscanf** or **textscan**.
- Store groups of related variables in separate MAT-files.

# Practice

Load Sample\_Text\_data.txt into variables **a**, **b**, **c**, and **d**.

- Do not suppress the command line output ("**;**") so that your import shows up when you publish your script.
- The import wizard is not an acceptable import method for this exercise.

Load Sample\_Text\_data.csv into variables **Date**, **Name**, and **Score**.

- Do not suppress the command line output ("**;**") so that your import shows up when you publish your script.
- Note that the data is separated by commas (i.e. Comma Separated Variables or .csv).
- The import wizard is not an acceptable import method for this exercise.

Write variables generated during the previous step (i.e. **Date**, **Name**, and **Score**) to an Excel files (you choose the filename).

# Practice problem

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  
x 2.2 y 6  
x 3.4 y 7.4
```

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 string. 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.

# Practice problem

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.

# Practice problem

For a biomedical experiment, the names and weights of some patients have been stored in a file 'patwts.dat'. For example, the file might look like this:

```
Darby George 166.2  
Helen Dee 143.5  
Giovanni Lupa 192.4  
Cat Donovan 215.1
```

Create this data file first. Then, write a script `readpatwts` that will first attempt to open the file. If the file open is not successful, an error message should be printed. If it is successful, the script will read the data into strings, one line at a time. Print for each person the name in the form 'last, first' followed by the weight. Also, calculate and print the average weight. Finally, print whether or not the file close was successful. For example, the result of running the script would look like this:

```
>> readpatwts  
George,Darby 166.2  
Dee,Helen 143.5  
Lupa,Giovanni 192.4  
Donovan,Cat 215.1  
The ave weight is 179.30  
File close successful
```

# Practice problem

Create a file 'parts\_inv.dat' that stores on each line a part number, cost, and quantity in inventory, for example:

```
123 5.99 52  
456 3.97 100  
333 2.22 567
```

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

# Practice problem

Create a file that 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  
f 1.1  
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.



# Practice problem

Create a file, 'phonenos.dat', of phone numbers in the following form:

```
6012425932  
6178987654  
8034562468
```

Read the phone numbers from the file and print them in the form:

```
601-242-5932
```

Use **load** to read the phone numbers.

# Practice problem

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.

# Practice problem

Create a data file that has points in a three-dimensional space stored in the following format:

```
x 2.2 y 5.3 z 1.8
```

Do this by creating *x*, *y*, and *z* vectors and then use **fprintf** to create the file in the specified format.

# Practice problem

A software package writes data to a file in a format that includes curly braces around each line and commas separating the values. For example, a data file `mm.dat` might look like this:

```
{33, 2, 11}  
{45, 9, 3}
```

Use the **fgetl** function in a loop to read this data. Create a matrix that stores just the numbers, and write the matrix to a new file. Assume that there are the same number of numbers in each line of the original file.

# Practice problem

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

```
Aerospace 201  
Civil 45  
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  
Civi 45  
Mech 66
```

# Practice problem

A software package writes data to a file in a format that includes curly braces around each line and commas separating the values. For example, a data file `mm.dat` might look like this:

```
{33, 2, 11}  
{45, 9, 3}
```

Use the **fgetl** function in a loop to read this data. Create a matrix that stores just the numbers, and write the matrix to a new file. Assume that there are the same number of numbers in each line of the original file.

# Practice problem

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.

# Practice problem

A spreadsheet, 'popdata.xls', stores the population every 20 years for a small town that underwent a boom and then a 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