Introduction to MATLAB

A. Prof./ Ahmed Nagib Elmekawy

Mechanical Engineering department, Alexandria University, Egypt

Spring 2023

Lecture 5

- **Functions**
- **Writing and reading to/from command window and files**
- **Interpolation**
- **Programming Errors and Best Practices**

Functions

Getting Help for Functions

You can use the `lookfor` command to find functions that are relevant to your application.

For example, type `lookfor imaginary` to get a list of the functions that deal with imaginary numbers. You will see listed:

```
i      Imaginary unit
j      Imaginary unit
imag   Complex imaginary part
```

Common mathematical functions:

Exponential

`exp(x)`	Exponential; e^x
`sqrt(x)`	Square root; $\sqrt{x}$

Logarithmic

`log(x)`	Natural logarithm; ln x
`log10(x)`	Common (base 10) logarithm; `log10` $x = \log_{10} x$

(continued…)

Some common mathematical functions (continued)

Complex

`abs(x)`	Absolute value.
`angle(x)`	Angle of a complex number.
`conj(x)`	Complex conjugate.
`imag(x)`	Imaginary part of a complex number.
`real(x)`	Real part of a complex number.

(continued…)

Some common mathematical functions (continued)

Numeric

`ceil(x)`	Round to nearest integer toward ∞.
`fix(x)`	Round to nearest integer toward zero.
`floor(x)`	Round to nearest integer toward -∞.
`round(x)`	Round toward nearest integer.
`sign(x)`	Signum function: +1 if $x > 0$; 0 if $x = 0$; -1 if $x < 0$.

Example

```
1 —    clc;clear
2 —    y=[2.3,2.6,3.9]
3 —    x1=round(y)
4 —    x2=fix(y)
5 —    x3=ceil(y)
6 —    z = [-2.6,-2.3,5.7];
7 —    x4=fix(z)
8 —    x5=floor(z)
```

`ceil(x)`	Round to nearest integer toward ∞.
`fix(x)`	Round to nearest integer toward zero.
`floor(x)`	Round to nearest integer toward -∞.
`round(x)`	Round toward nearest integer.

Example

In Command Window

```
y= 2.3000   2.6000   3.9000
x1 =
     2     3     4
x2 =
     2     2     3
x3 =
     3     3     4
z =
   -2.6000   -2.3000    5.7000
x4 =
    -2    -2     5
x5 =
    -3    -3     5
```

```
1 —    clc;clear
2 —    y=[2.3,2.6,3.9];
3 —    x1=round(y)
4 —    x2=fix(y)
5 —    x3=ceil(y)
6 —    z = [-2.6,-2.3,5.7];
7 —    x4=fix(z)
8 —    x5=floor(z)
```

`ceil(x)`	Round to nearest integer toward ∞.
`fix(x)`	Round to nearest integer toward zero.
`floor(x)`	Round to nearest integer toward -∞.
`round(x)`	Round toward nearest integer.

Operations on Arrays

MATLAB will treat a variable as an array automatically. For example, to compute the square roots of 5, 7, and 15, type

```
>>x = [5,7,15];
>>y = sqrt(x)
y =
   2.2361    2.6358    3.8730
```

Expressing Function Arguments

We can write sin 2 in text, but MATLAB requires parentheses surrounding the 2 (which is called the *function argument* or *parameter*).

Thus to evaluate sin 2 in MATLAB, we type `sin(2)`. The MATLAB function name must be followed by a pair of parentheses that surround the argument.

To express in text the sine of the second element of the array *x*, we would type `sin[x(2)]`. However, in MATLAB you cannot use square brackets or braces in this way, and you must type `sin(x(2))`.

(continued …)

Expressing Function Arguments (continued)

To evaluate $\sin(x^2 + 5)$, you type `sin(x.^2 + 5)`.

To evaluate $\sin(\sqrt{x} + 1)$, you type `sin(sqrt(x)+1)`.

Using a function as an argument of another function is called *function composition.* Be sure to check the order of precedence and the number and placement of parentheses when typing such expressions.

Every left-facing parenthesis requires a right-facing mate. However, this condition does not guarantee that the expression is correct!

Expressing Function Arguments (continued)

Another common mistake involves expressions like $\sin^2 x$, which means $(\sin x)^2$.

In MATLAB we write this expression as `(sin(x))^2`, *not* as `sin^2(x)`, `sin^2x`, `sin(x^2)`, or `sin(x)^2`!

Expressing Function Arguments (continued)

The MATLAB trigonometric functions operate in radian mode. Thus `sin(5)` computes the sine of 5 rad, not the sine of 5°.

To convert between degrees and radians, use the relation $q_{\text{radians}} = (pi\ /180)\ q_{\text{degrees}}$.

Trigonometric functions: Table 3.1–2, page 116

`cos(x)`	Cosine; cos *x*.
`cot(x)`	Cotangent; cot *x*.
`csc(x)`	Cosecant; csc *x*.
`sec(x)`	Secant; sec *x*.
`sin(x)`	Sine; sin *x*.
`tan(x)`	Tangent; tan *x*.

Inverse Trigonometric functions: Table 3.1–2

`acos(x)`	Inverse cosine; arccos *x*.
`acot(x)`	Inverse cotangent; arccot *x*.
`acsc(x)`	Inverse cosecant; arccsc *x*.
`asec(x)`	Inverse secant; arcsec *x*.
`asin(x)`	Inverse sine; arcsin *x* .
`atan(x)`	Inverse tangent; arctan *x* .
`atan2(y,x)`	Four-quadrant inverse tangent.

Hyperbolic functions: Table 3.1–3, page 119

`cosh(x)`	Hyperbolic cosine
`coth(x)`	Hyperbolic cotangent.
`csch(x)`	Hyperbolic cosecant
`sech(x)`	Hyperbolic secant
`sinh(x)`	Hyperbolic sine
`tanh(x)`	Hyperbolic tangent

Inverse Hyperbolic functions: Table 3.1–3

`acosh(x)`	Inverse hyperbolic cosine
`acoth(x)`	Inverse hyperbolic cotangent
`acsch(x)`	Inverse hyperbolic cosecant
`asech(x)`	Inverse hyperbolic secant
`asinh(x)`	Inverse hyperbolic sine
`atanh(x)`	Inverse hyperbolic tangent;

User-Defined Functions

The first line in a function file must begin with a *function definition line* that has a list of inputs and outputs. This line distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = name(input variables)
```

Note that the output variables are enclosed in *square brackets,* while the input variables must be enclosed with *parentheses.* The function name (here, `name`) should be the same as the file name in which it is saved (with the .m extension).

More? See pages 119-123.

User-Defined Functions: Example

```
function z = fun(x,y)
u = 3*x;
z = u + 6*y.^2;
```

Note the use of a semicolon at the end of the lines. This prevents the values of `u` and `z` from being displayed.

Note also the use of the array exponentiation operator (`.^`). This enables the function to accept `y` as an array.

(continued …)

User-Defined Functions: Example (continued)

Call this function with its output argument:

```
>>z = fun(3,7)
z =
    303
```

The function uses x = 3 and y = 7 to compute z.

(continued …)

User-Defined Functions: Example (continued)

Call this function without its output argument and try to access its value. You will see an error message.

```
>>fun(3,7)
ans =
     303
>>z
???  Undefined function or variable 'z'.
```

(continued …)

User-Defined Functions: Example (continued)

Assign the output argument to another variable:

```
>>q = fun(3,7)
q =
   303
```

You can suppress the output by putting a semicolon after the function call.

For example, if you type `q = fun(3,7);` the value of `q` will be computed but not displayed (because of the semicolon).

Local Variables: The variables `x` and `y` are *local* to the function `fun`, so unless you pass their values by naming them `x` and `y`, their values will not be available in the workspace outside the function. The variable `u` is also local to the function. For example,

```
>>x = 3;y = 7;
>>q = fun(x,y);
>>x
x =
  3
>>y
y =
  7
>>z
??? Undefined function or variable 'z'.
```

Only the order of the arguments is important, not the names of the arguments:

```
>>x = 7;y = 3;
>>z = fun(y,x)
z =
   303
```

The second line is equivalent to `z = fun(3,7)`.

You can use arrays as input arguments:

```
>>r = fun(2:4,7:9)
r =
   300    393    498
```

A function may have more than one output. These are enclosed in square brackets.

For example, the function `circle` computes the area *A* and circumference *C* of a circle, given its radius as an input argument.

```
function [A, C] = circle(r)
A = pi*r.^2;
C = 2*pi*r;
```

The function is called as follows, if the radius is 4.

```
>>[A, C] = circle(4)
A =
  50.2655
C =
  25.1327
```

A function may have no input arguments and no output list.

For example, the function `show_date` clears all variables, clears the screen, computes and stores the date in the variable `today`, and then displays the value of `today`.

```
function show_date
clear
clc
today = date
```

Examples of Function Definition Lines

1. One input, one output:

```
function [area_square] = square(side)
```

2. Brackets are optional for one input, one output:

```
function area_square = square(side)
```

3. Two inputs, one output:

```
function [volume_box] = box(height,width,length)
```

4. One input, two outputs:

```
function [area_circle,circumf] = circle(radius)
```

5. No named output: `function sqplot(side)`

Function Example

```
function [dist,vel] = drop(g,v0,t);
% Computes the distance travelled and the
% velocity of a dropped object,
% as functions of g,
% the initial velocity v0, and
% the time t.
vel = g*t + v0;
dist = 0.5*g*t.^2 + v0*t;
```

(continued …)

Function Example (continued)

1. The variable names used in the function definition may, but need not, be used when the function is called:

```
>>a = 32.2;
>>initial_speed = 10;
>>time = 5;
>>[feet_dropped,speed] = . . .
drop(a,initial_speed,time)
```

(continued …)

Function Example (continued)

2. The input variables need not be assigned values outside the function prior to the function call:

```
[feet_dropped,speed] = drop(32.2,10,5)
```

3. The inputs and outputs may be arrays:

```
[feet_dropped,speed]=drop(32.2,10,0:1:5)
```

This function call produces the arrays `feet_dropped` and `speed`, each with six values corresponding to the six values of time in the array `time`.

Local Variables

The names of the input variables given in the function definition line are local to that function.

This means that other variable names can be used when you call the function.

All variables inside a function are erased after the function finishes executing, except when the same variable names appear in the output variable list used in the function call.

Global Variables

The `global` command declares certain variables global, and therefore their values are available to the basic workspace and to other functions that declare these variables global.

The syntax to declare the variables `a`, `x`, and `q` is

```
global a x q
```

Any assignment to those variables, in any function or in the base workspace, is available to all the other functions declaring them global.

Writing and reading to/from command window and files

Importing data from command window

- The variable is defined in the script file, but a specific value is entered in the Command Window when the script file is executed.
- The variable is defined in the script file, and when the file is executed, the user is prompted to assign a value to the variable in the Command Window. This is done by using the input command for creating the variable. The form of the input command is:

```
variable_name = input('string with a message that
                      is displayed in the Command Window')
```

Importing data from command window

M-file

```
% This script file calculates the average of points scored in
three games.
% The points from each game are assigned to the variables by
% using the input command.
game1=input('Enter the points scored in the first game ');
game2=input('Enter the points scored in the second game ');
game3=input('Enter the points scored in the third game ');
ave_points=(game1+game2+game3)/3
```

In command Window

```
Enter the points scored in the first game   67
Enter the points scored in the second game   91
Enter the points scored in the third game   70

ave_points =
    76
```

The computer displays the message. Then the value of the score is typed by the user and the **Enter** key is pressed.

Importing data from command window

- The `input` command can also be used to assign a string to a variable.

```
variable_name = input('prompt message','s')
```

Output Commands
The disp command

- The `disp` command is used to display the elements of a variable without displaying the name of the variable, and to display text. The format of the `disp` command is:

```
disp(name of a variable) or disp('text as string')
```

- Every time the `disp` command is executed, the display it generates appears in a new line. One example is:

```
>> abc = [5  9  1;  7  2  4];
>> disp(abc)
     5     9     1
     7     2     4

>> disp('The problem has no solution.')

The problem has no solution.
>>
```

A 2×3 array is assigned to variable abc.

The disp command is used to display the abc array.

The array is displayed without its name.

The disp command is used to display a message.

Example
Matlab Code

```
% This script file calculates the average points scored in three games.
% The points from each game are assigned to the variables by
% using the input command.
% The disp command is used to display the output.

game1=input('Enter the points scored in the first game    ');
game2=input('Enter the points scored in the second game   ');
game3=input('Enter the points scored in the third game    ');
ave_points=(game1+game2+game3)/3;
disp(' ')
disp('The average of points scored in a game is:')
disp(' ')
disp(ave_points)
```

Display empty line.

Display text.

Display empty line.

Display the value of the variable `ave_points`.

In Command Window

```
Enter the points scored in the first game    89
Enter the points scored in the second game   60
Enter the points scored in the third game    82

The average of points scored in a game is:

   77
```

An empty line is displayed.

The text line is displayed.

An empty line is displayed.

The value of the variable `ave_points` is displayed.

Output Commands
The disp command

- In many situations it is nice to display output (numbers) in a table. This can be done by first defining a variable that is an array with the numbers and then using the `disp` command to display the array Headings to the columns can also be created with the `disp` command. Since in the `disp` command the user cannot control the format (the width of the columns and the distance between the columns) of the display of the array, the position of the headings has to be aligned with the columns by adding spaces.

Example
Matlab Code

```
yr=[1984 1986 1988 1990 1992 1994 1996];
pop=[127 130 136 145 158 178 211];
tableYP(:,1)=yr';
tableYP(:,2)=pop';
disp('         YEAR      POPULATION')
disp('                   (MILLIONS)')
disp(' ')
disp(tableYP)
```

The population data is entered in two row vectors.

yr is entered as the first column in the array tableYP.

pop is entered as the second column in the array tableYP.

Display heading (first line).

Display heading (second line).

Display an empty line.

Display the array tableYP.

In Command Window

```
        YEAR      POPULATION
                  (MILLIONS)

        1984         127
        1986         130
        1988         136
        1990         145
        1992         158
        1994         178
        1996         211
```

Headings are displayed.

An empty line is displayed.

The tableYP array is displayed.

The `fprintf` command

Using the `fprintf` command to display text:

Display text with

```
fprintf('Text to display')
```

Example

```
>> fprintf( 'Howdy neighbor' )
Howdy neighbor>>
```

Problem – Command Window displays prompt (>>) at end of text, not at start of next line!

The `fprintf` command

To make the next thing that MATLAB writes (after a use of `fprintf`) appear on the start of a new line, put the two characters "\n" at the end of the `fprintf` text

```
>> fprintf( 'Howdy neighbor\n' )
Howdy neighbor
>>
```

The `fprintf` command

Can also use `\n` in middle of text to make MATLAB display remainder of text on next line

```
>> fprintf('A man\n A plan\n Panama\n')
A man
A plan
A canal
Panama
>>
```

The `fprintf` command

`\` is an *escape character*, a special combination of two characters that makes `fprintf` do something instead of print the two characters

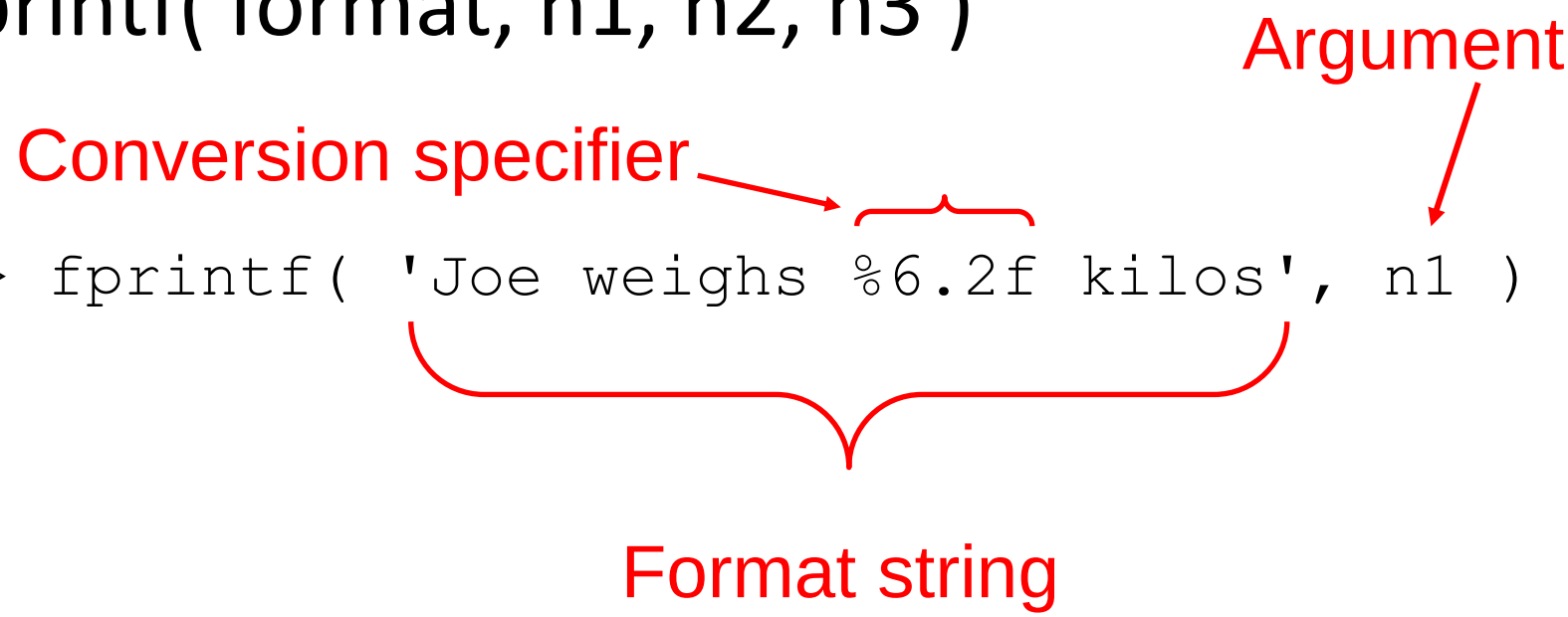
`\n` – makes following text come out at start of next line

`\t` – horizontal tab

There are a few more

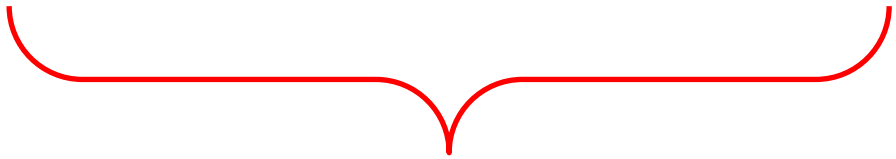
The `fprintf` command

fprintf(format, n1, n2, n3)



The `fprintf` command

```
>> fprintf( 'Joe weighs %6.2f kilos', n1 )
```



Format string

- May contain text and/or conversion specifiers
- Must be enclosed in SINGLE quotes, not double quotes, aka quotation marks (" ")

The `fprintf` command

```
>> fprintf( 'Joe is %d weighs %f kilos', age, weight )
```

Arguments

- Number of arguments and conversion specifiers must be the same
- Leftmost conversion specifier formats leftmost argument, 2nd to left specifier formats 2nd to left argument, etc.

The `fprintf` command

Conversion specifier

```
>> fprintf( 'Joe weighs %f kilos', n1 )
```

Common conversion specifiers

- %f fixed point (decimal always between 1's and 0.1's place,
  e.g., 3.14, 56.8
- %e scientific notation, e.g, 2.99e+008
- %d integers (no decimal point shown)
- %s string of characters

The `fprintf` command

Conversion specifier

```
>> fprintf( 'Joe weighs %6.2f kilos', n1 )
```

To control display in fixed or scientific, use `%w.pf` or `%w.pe`

- w = width: the minimum number of characters to be displayed
- p = “precision”: the number of digits to the right of the decimal point

If you omit "w", MATLAB will display correct precision and just the right length

The `fprintf` command

```
>> e = exp( 1 );
>> fprintf( 'e is about %4.1f\n', e )
e is about  2.7
>> fprintf( 'e is about %10.8f\n', e )
e is about 2.71828183
>> fprintf( 'e is about %10.8e', e )
e is about 2.71828183e+000
>> fprintf( 'e is about %10.2e', e )
e is about  2.72e+000
>> fprintf( 'e is about %f\n', e )
e is about 2.718282
```

The `fprintf` command

Use escape characters to display characters used in conversion specifiers

- To display a percent sign, use %% in the text
- To display a single quote, use '' in the text (two sequential single quotes)
- To display a backslash, use \\ in the text (two sequential backslashes)

The `fprintf` command

Make the following strings

- Mom's apple 3.14
- Mom's apple 3.1415926
- Mom's apple 3.1e+000

```
>> fprintf( 'Mom''s apple %.2f\n', pi )
Mom's apple 3.14
>> fprintf( 'Mom''s apple %.7f\n', pi )
Mom's apple 3.1415927
>> fprintf( 'Mom''s apple %.1e\n', pi )
Mom's apple 3.1e+000
```

The `fprintf` command

Format strings are often long. Can break a string by

1. Put an open square bracket ([ ) in front of first single quote
2. Put a second single quote where you want to stop the line
3. Follow that quote with an ellipsis (three periods)
4. Press ENTER, which moves cursor to next line
5. Type in remaining text in single quotes
6. Put a close square bracket ( ])
7. Put in the rest of the `fprintf` command

The `fprintf` command

Example

```
>> weight = 178.3;
>> age = 17;
>> fprintf( ['Tim weighs %.1f lbs'...
' and is %d years old'], weight, age )

 Tim weighs 178.3 lbs and is 17 years old
```

The `fprintf` command

`fprintf` is *vectorized*, i.e., when vector or matrix in arguments, command repeats until all elements displayed

- Uses matrix data column by column

```
x=1:5;
y=sqrt(x);
T=[x; y]
fprintf('If the number is: %i, its square root is: %f\n',T)
```

Create a vector x.

Create a vector y.

Create 2 × 5 matrix T, first row is x, second row is y.

The `fprintf` command displays two numbers from T in every line.

The `fprintf` command

When this script file is executed the display in the Command Window is:

```
T =
    1.0000    2.0000    3.0000    4.0000    5.0000
    1.0000    1.4142    1.7321    2.0000    2.2361
If the number is: 1, its square root is: 1.000000
If the number is: 2, its square root is: 1.414214
If the number is: 3, its square root is: 1.732051
If the number is: 4, its square root is: 2.000000
If the number is: 5, its square root is: 2.236068
```

The 2×5 matrix `T`.

The `fprintf` command repeats 5 times, using the numbers from the matrix `T` column after column.

The `fprintf` command

Using the `fprintf` command to save output to a file:

Takes three steps to write to a file

Step a: – open file

```
fid=fopen('file_name','permission')
```

`fid` – *file identifier*, lets `fprintf` know what file to write its output in

`permission` – tells how file will be used, e.g., for reading, writing, both, etc.

The `fprintf` command

Some common permissions

- `r` - open file for reading
- `w` - open file for writing. If file exists, content deleted. If file doesn't exist, new file created
- `a` - same as `w` except if file exists the written data is appended to the end of the file
- If no permission code specified, `fopen` uses `r`

See Help on `fopen` for all permission codes

The `fprintf` command

Step b:

Write to file with `fprintf`. Use it exactly as before but insert `fid` before the format string, i.e.,

```
fprintf(fid,'format string',variables)
```

The passed `fid` is how `fprintf` knows to write to the file instead of display on the screen

The `fprintf` command

Step c:

When you're done writing to the file, close it with the command `fclose(fid)`

- Once you close it, you can't use that `fid` anymore until you get a new one by calling `fopen`

Make sure to close every file you open. Too many open files makes problems for MATLAB

The `fprintf` command

Miscellaneous

- If the file name you give to `fopen` has no path, MATLAB writes it to the current directory, also called the working directory
- You can have multiple files open simultaneously and use `fprintf` to write to all of them just by passing it different `fid`s
- You can read the files you make with `fprintf` in any text editor, e.g., MATLAB's Editor window or Notepad

Example

```
% Script file in which fprintf is used to write output to files.
% Two conversion tables are created and saved to two different files.
% One converts mi/h to km/h, the other converts lb to N.
clear all
Vmph=10:10:100;
Vkmh=Vmph.*1.609;
TBL1=[Vmph; Vkmh];
Flb=200:200:2000;
FN=Flb.*4.448;
TBL2=[Flb; FN];
fid1=fopen('VmphtoVkm.txt','w');
fid2=fopen('FlbtoFN.txt','w');
fprintf(fid1,'Velocity Conversion Table\n \n');
fprintf(fid1,'      mi/h          km/h     \n');
fprintf(fid1,'   %8.2f       %8.2f\n',TBL1);
fprintf(fid2,'Force Conversion Table\n \n');
fprintf(fid2,'     Pounds        Newtons     \n');
fprintf(fid2,'   %8.2f       %8.2f\n',TBL2);
fclose(fid1);
fclose(fid2);
```

Creating a vector of velocities in mi/h.

Converting mph to km/h.

Creating a table (matrix) with two rows.

Creating a vector of forces in lb.

Converting lb to N.

Creating a table (matrix) with two rows.

Open a .txt file named VmphtoVkm.

Open a .txt file named FlbtoFN.

Writing a title and an empty line to the file fid1.

Writing two column headings to the file fid1.

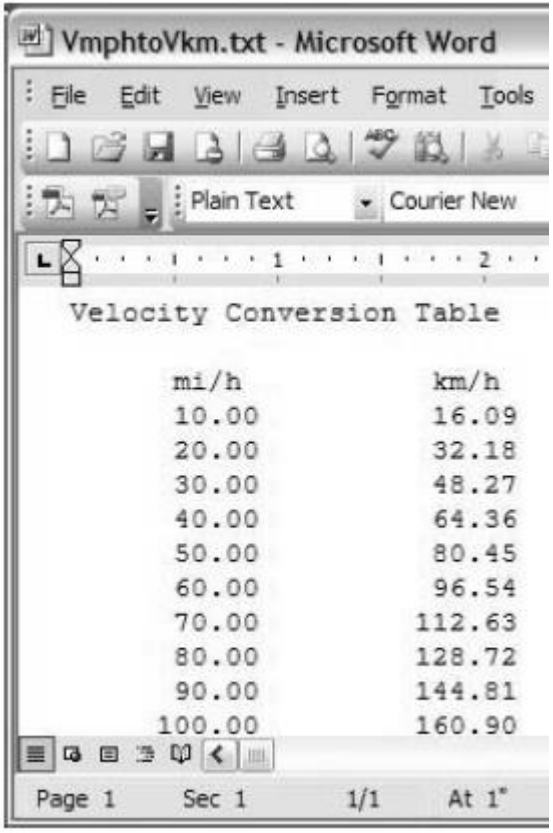
Writing the data from the variable TBL1 to the file fid1.

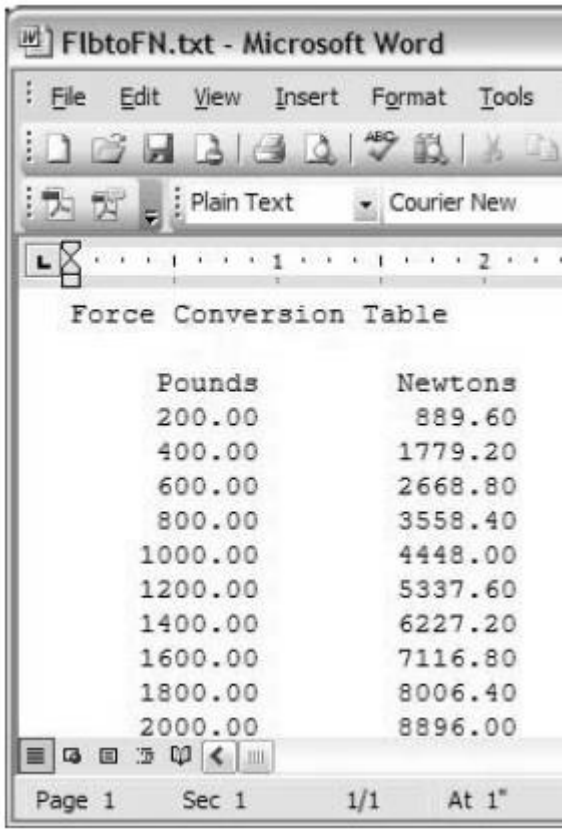
Writing the force conversion table (data in variable TBL2) to the file fid2.

Closing the files fid1 and fid2.

Example

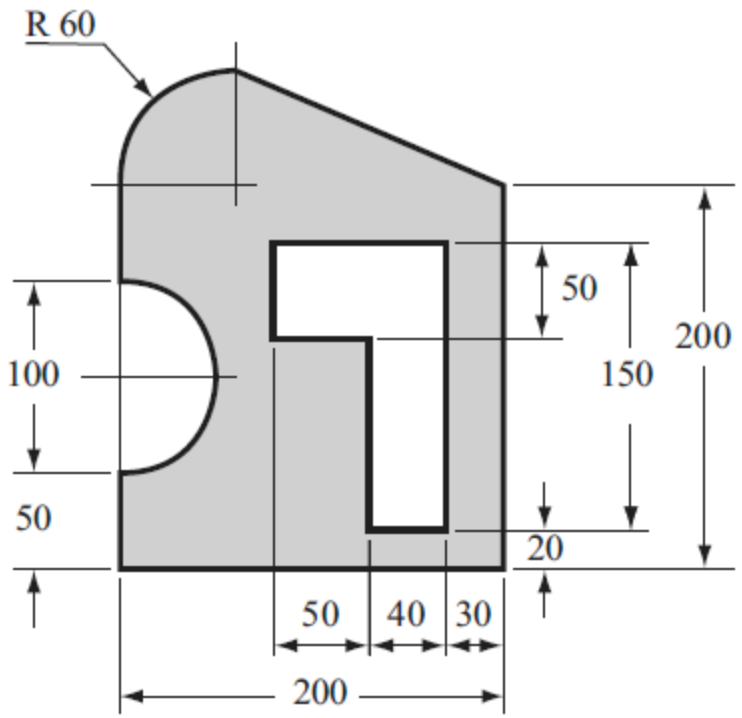
When the script file above is executed two new .txt files, named VmphtoVkm and FlbtoFN, are created and saved in the current directory. These files can be opened with any application that can read .txt files.





Example: Centroid of a composite area

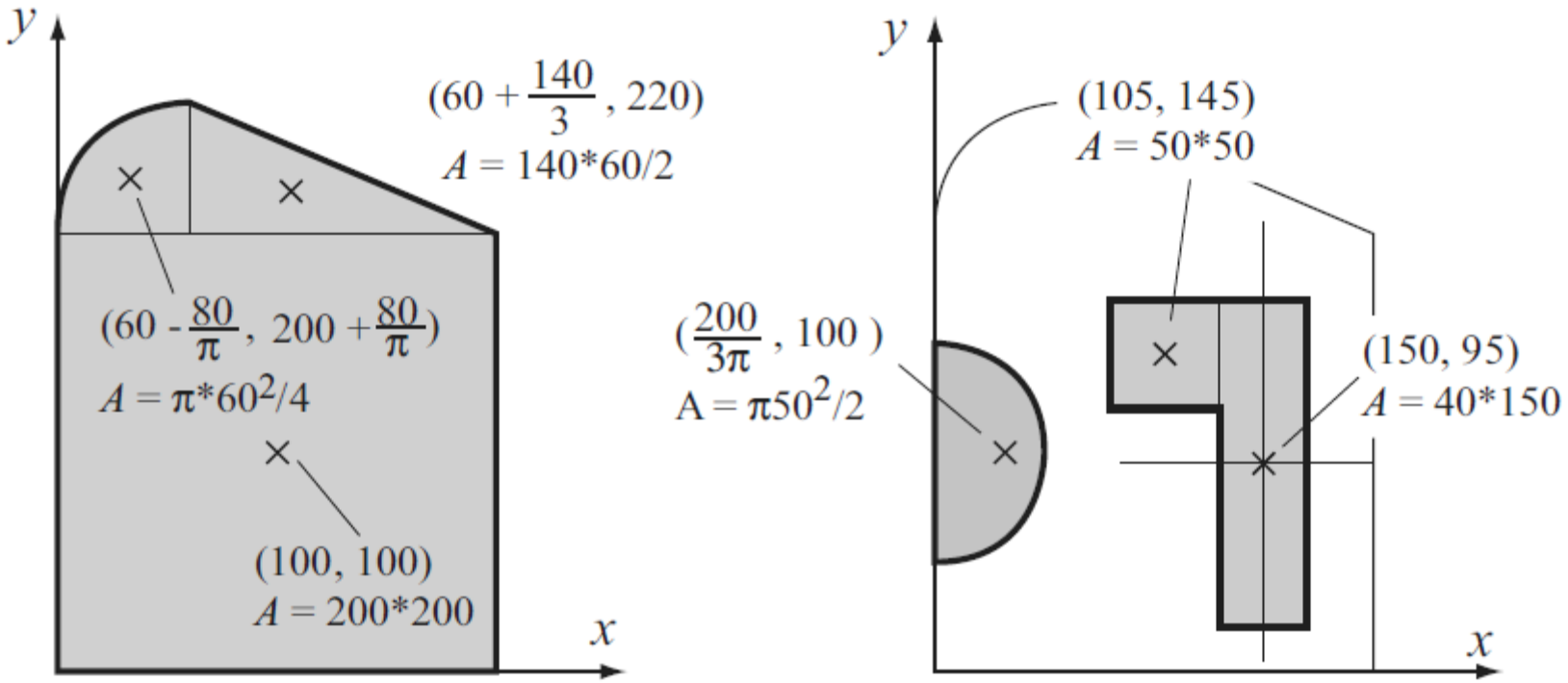
Write a program in a script file that calculates the coordinates of the centroid of a composite area. (A composite area can easily be divided into sections whose centroids are known.) The user needs to divide the area into sections and know the coordinates of the centroid (two numbers) and the area of each section (one number). When the script file is executed, it asks the user to enter the three numbers as a row in a matrix. The user enters as many rows as there are sections. A section that represents a hole is taken to have a negative area. For output, the program displays the coordinates of the centroid of the composite area. Use the program to calculate the centroid of the area shown in the figure.



Dimensions in mm

Example: Centroid of a composite area

The area is divided into six sections as shown in the following figure. The total



Units: coordinates mm, area mm^2

Example: Centroid of a composite area

```
% The program calculates the coordinates of the centroid
% of a composite area.
clear C xs ys As
C=input('Enter a matrix in which each row has three ele-
ments.\nIn each row enter the x and y coordinates of the
centroid and the area of a section.\n');
xs=C(:,1)';
ys=C(:,2)';
As=C(:,3)';
A=sum(As);
x=sum(As.*xs)/A;
y=sum(As.*ys)/A;
fprintf('The coordinates of the centroid are: ( %f, %f )\n',x,y)
```

Creating a row vector for the x coordinate of each section (first column of C).

Creating a row vector for the y coordinate of each section (second column of C).

Creating a row vector for the area of each section (third column of C).

Calculating the total area.

Calculating the coordinates of the centroid of the composite area.

Example: Centroid of a composite area

Command Window

```
Enter a matrix in which each row has three elements.

In each row enter the x and y coordinates of the centroid
and the area of a section.
[100 100 200*200
60-80/pi 200+80/pi pi*60^2/4
60+140/3 220 140*60/2
200/(3*pi) 100 -pi*50^2/2
105 145 -50*50
150 95 -40*150]

The coordinates of the centroid are: ( 85.387547 , 131.211809 )
```

Entering the data for matrix C. Each row has three elements: the x, y, and A of a section.

Importing Spreadsheet Files

Importing data from Excel is done with the `xlsread` command. When the command is executed, the data from the spreadsheet is assigned as an array to a variable.

```
variable_name=xlsread('filename')
```

- `'filename'` (typed as a string) is the name of the Excel file. The directory of the Excel file must be either the current directory or listed in the search path.
- If the Excel file has more than one sheet, the data will be imported from the first sheet.

Importing Spreadsheet Files

When an Excel file has several sheets, the `xlsread` command can be used to import data from a specified sheet. The form of the command is then:

```
variable_name = xlsread('filename','sheet_name')
```

- The name of the sheet is typed as a string.

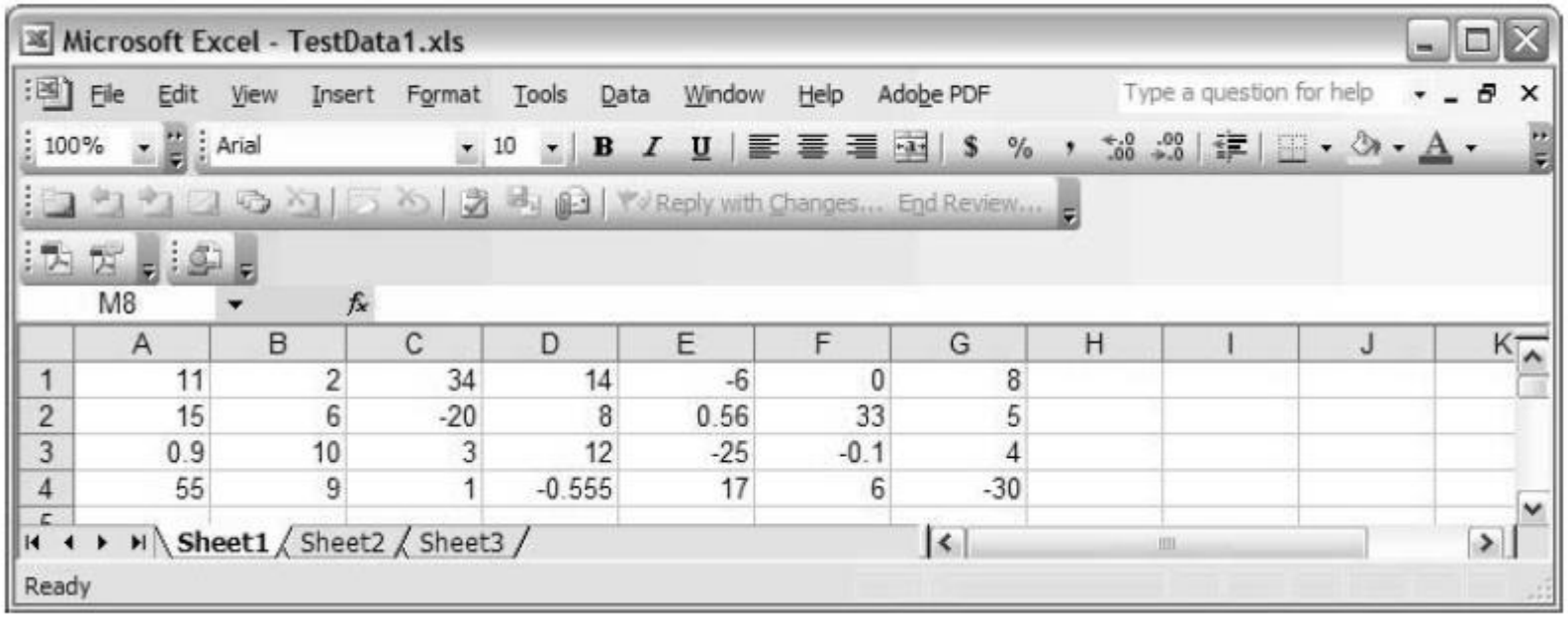
Another option is to import only a portion of the data that is in the spreadsheet. This is done by typing an additional argument in the command:

```
variable_name = xlsread('filename','sheet_name','range')
```

- The `'range'` (typed as a string) is a rectangular region of the spreadsheet defined by the addresses (in Excel notation) of the cells at opposite corners of the region. For example, `'C2:E5'` is a region of rows 2, 3, 4, and 5 and columns *C*, *D*, and *E*.

Example

The data from the Excel spreadsheet shown in Figure is imported into MATLAB by using the `xlsread` command



	A	B	C	D	E	F	G	H	I	J	K
1	11	2	34	14	-6	0	8				
2	15	6	-20	8	0.56	33	5				
3	0.9	10	3	12	-25	-0.1	4				
4	55	9	1	-0.555	17	6	-30				

```
>> DATA = xlsread('TestData1')

DATA =

 11.0000   2.0000  34.0000  14.0000  -6.0000         0   8.0000
 15.0000   6.0000 -20.0000   8.0000   0.5600  33.0000   5.0000
  0.9000  10.0000   3.0000  12.0000 -25.0000  -0.1000   4.0000
 55.0000   9.0000   1.0000  -0.5550  17.0000   6.0000 -30.0000
```

Importing Spreadsheet Files

The command `A = xlsread('filename')` imports the Microsoft Excel workbook file `filename.xls` into the array `A`. The command `[A, B] = xlsread('filename')` imports all numeric data into the array `A` and all text data into the cell array `B`.

Exporting Data to Spreadsheet Files

Exporting data from MATLAB to an Excel spreadsheet is done by using the `xlswrite` command. The simplest form of the command is:

```
xlswrite('filename',variable_name)
```

- `'filename'` (typed as a string) is the name of the Excel file to which the data is exported. The file must be in the current directory. If the file does not exist, a new Excel file with the specified name will be created.
- `variable_name` is the name of the variable in MATLAB with the assigned data that is being exported.
- The arguments `'sheet_name'` and `'range'` can be added to the `xlswrite` command to export to a specified sheet and to a specified range of cells, respectively.

Interpolation

Interpolation

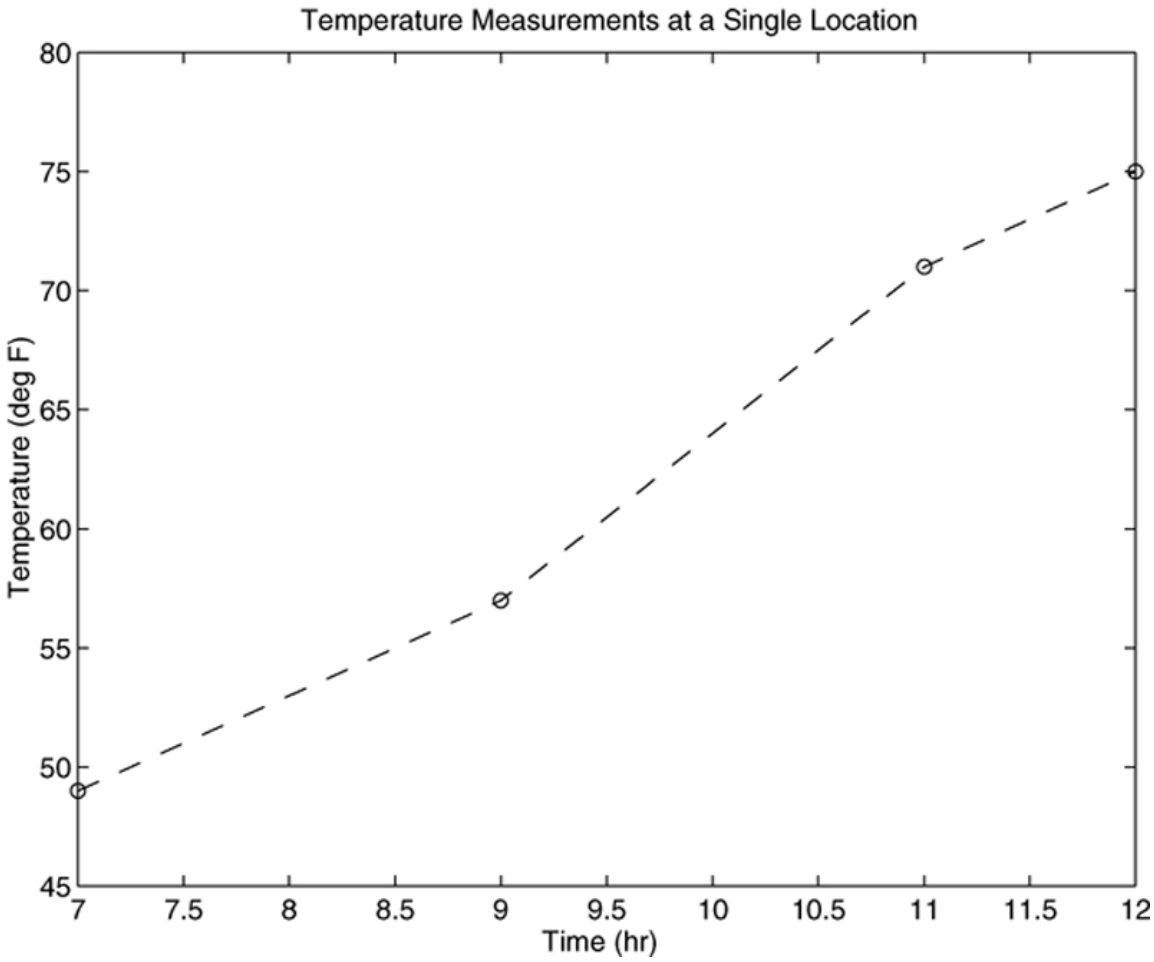
- In some applications we want to estimate a variable's value between the data points. This process is called interpolation. In other cases we might need to estimate the variable's value outside the given data range. This process is called extrapolation. Interpolation and extrapolation are greatly aided by plotting the data.

- Suppose we have the following temperature measurements, taken once an hour starting at 7:00 A.M. The measurements at 8 and 10 A.M. are missing for some reason, perhaps because of equipment malfunction.

Time	7 A.M.	9 A.M.	11 A.M.	12 noon
Temperature (°F)	49	57	71	75

Applications of interpolation: A plot of temperature data versus time.

Time	7 A.M.	9 A.M.	11 A.M.	12 noon
Temperature (°F)	49	57	71	75



Interpolation

Matlab Code

```
x = [7, 9, 11, 12];
y = [49, 57, 71, 75];
x_int = [8, 10];
interp1(x,y,x_int)
```

In Command Window

```
ans =
53
64
```

Interpolation

- You must keep in mind two restrictions when using the interp1 function.
- The values of the independent variable in the vector x must be in ascending order
- The values in the interpolation vector `x_int` must lie within the range of the values in x. Thus we cannot use the interp1 function to estimate the temperature at 6 A.M., for example.

Interpolation

- The interp1 function can be used to interpolate in a table of values by defining y to be a matrix instead of a vector. For example, suppose that we now have temperature measurements at three locations and the measurements at 8 and 10 A.M. are missing for all three locations. The data are as follows:

	Temperature (°F)		
Time	Location 1	Location 2	Location 3
7 A.M.	49	52	54
9 A.M.	57	60	61
11 A.M.	71	73	75
12 noon	75	79	81

We define x as before, but now we de ne y to be a matrix whose three columns contain the second, third, and fourth columns of the preceding table.

Interpolation

Matlab Code

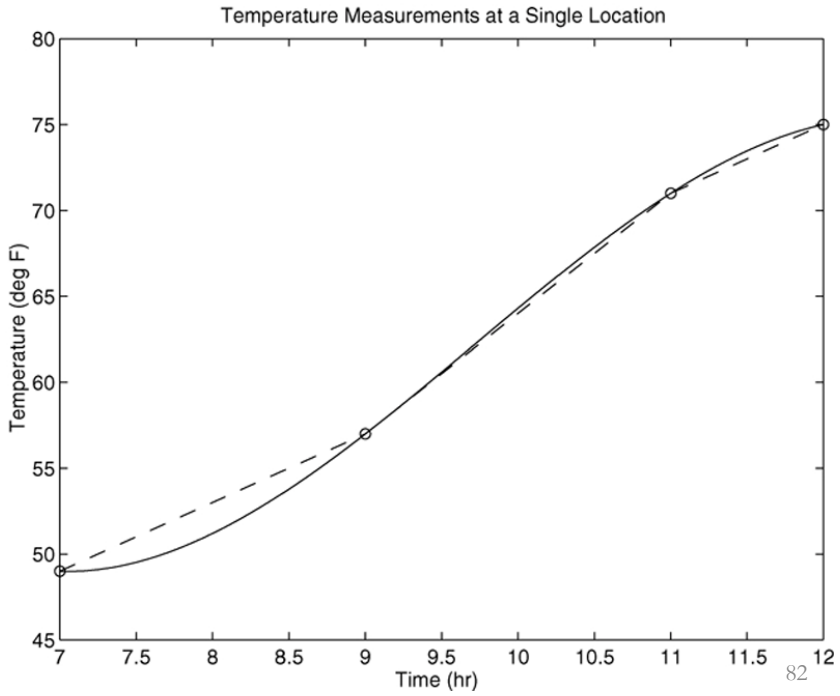
```
x = [7, 9, 11, 12]';
y(:,1) = [49, 57, 71, 75]';
y(:,2) = [52, 60, 73, 79]';
y(:,3) = [54, 61, 75, 81]';
x_int = [8, 10]';
interp1(x,y,x_int)
```

In Command Window

```
ans =
53.0000 56.0000 57.5000
64.0000 65.5000 68.0000
```

Cubic-spline interpolation

- High-order polynomials can exhibit undesired behavior between the data points, and this can make them unsuitable for interpolation. A widely used alternative procedure is to t the data points using a lower -order polynomial between each pair of adjacent data points. This method is called spline interpolation and is so named for the splines used by illustrators to draw a smooth curve through a set of points.



Cubic-spline interpolation

```
y_int = spline(x,y,x_int)
```

Computes a cubic-spline interpolation where `x` and `y` are vectors containing the data and `x_int` is a vector containing the values of the independent variable *x* at which we wish to estimate the dependent variable *y*. The result `Y_int` is a vector the same size as `x_int` containing the interpolated values of *y* that correspond to `x_int`.

Cubic-spline interpolation

- If the data are given as n pairs of (x, y) values, then n - 1 cubic polynomials are used. Each has the form

$$y_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

for $x_i \leq x \leq x_{i+1}$ and i =1,2, …., n

- The coefficients a_i, b_i, c_i, and d_i for each polynomial are determined so that the following three conditions are satisfied for each polynomial:
    1. The polynomial must pass through the data points at its endpoints at xi and xi!1.
    2. The slopes of adjacent polynomials must be equal at their common data point.
    3. The curvatures of adjacent polynomials must be equal at their common data point.

Cubic-spline interpolation

- For example, a set of cubic splines for the temperature data given earlier follows (y represents the temperature values, and x represents the hourly values). The data are repeated here.

x	7	9	11	12
y	49	57	71	75

- We will shortly see how to use MATLAB to obtain these polynomials.

For $7 \le x \le 9$

$$y_1(x) = -0.35(x - 7)^3 + 2.85(x - 7)^2 - 0.3(x - 7) + 49$$

For $9 \le x \le 11$

$$y_2(x) = -0.35(x - 9)^3 + 0.75(x - 9)^2 + 6.9(x - 9) + 57$$

For $11 \le x \le 12$

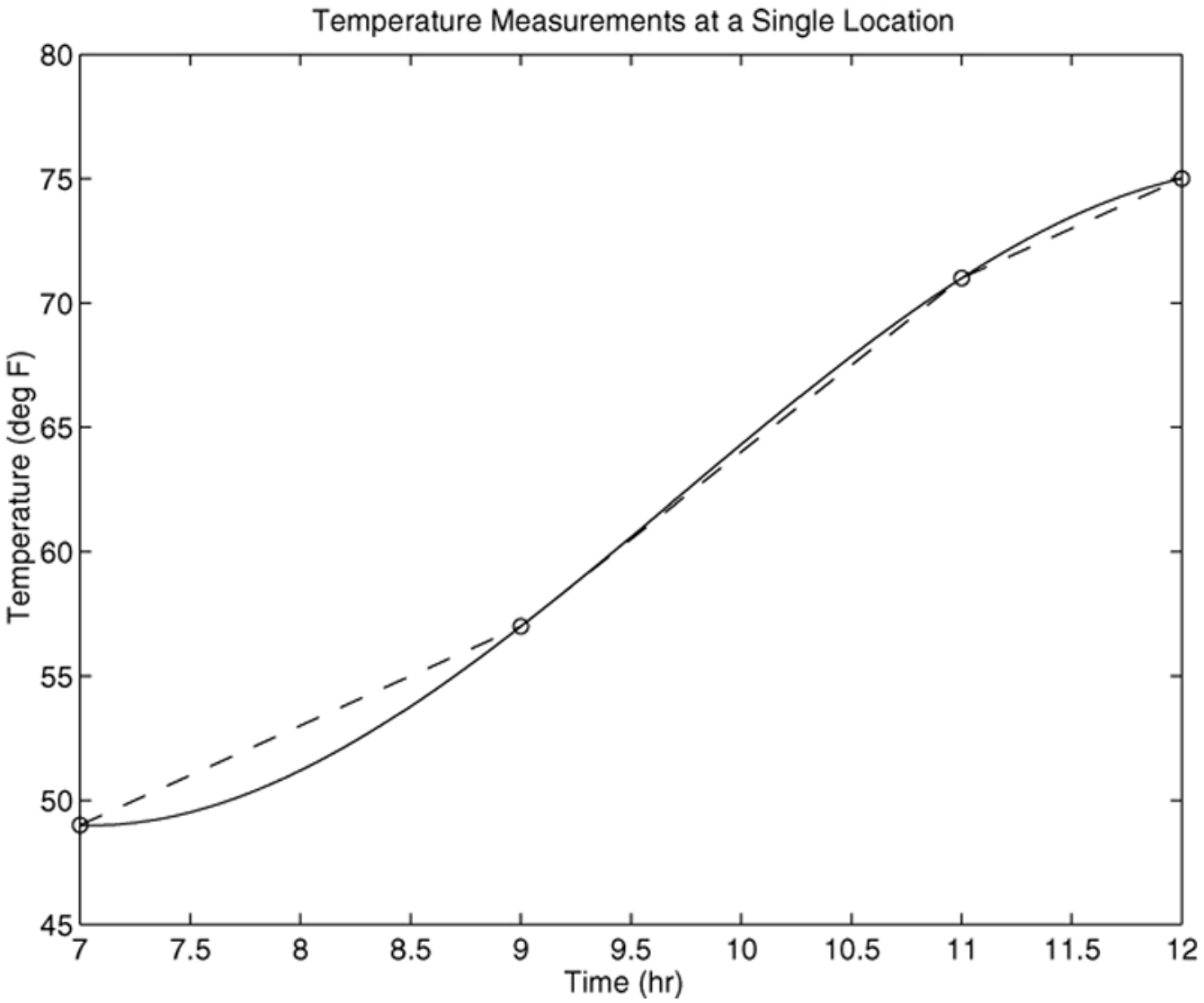
$$y_3(x) = -0.35(x - 11)^3 - 1.35(x - 11)^2 + 5.7(x - 11) + 71$$

Cubic-spline interpolation: Matlab Code

```
x = [7,9,11,12];
y = [49,57,71,75];
x_int = 7:0.01:12;
y_int = spline(x,y,x_int);
plot(x,y,'o',x,y,'--',x_int,y_int)
 xlabel('Time (hr)'),
 ylabel('Temperature (deg F)')
 title('Temperature Measurements at a ...
        Single Location')
 axis([7 12 45 80])
```

This produces the next figure.

Linear and cubic-spline interpolation of temperature data.



Polynomial interpolation functions.

Command

```
y_est = interp1(x,y,x_est,'spline')
```

Description

Returns a column vector `y_est` that contains the estimated values of *y* that correspond to the *x* values specified in the vector `x_est`, using cubic-spline interpolation.

Table 7.4–2 Continued

```
[breaks, coeffs, m, n] = unmkpp(spline(x,y))
```

Computes the coefficients of the cubic-spline polynomials for the data in `x` and `y`. The vector `breaks` contains the *x* values, and the matrix `coeffs` is an $m \times n$ matrix containing the polynomial coefficients. The scalars `m` and `n` give the dimensions of the matrix `coeffs`; `m` is the number of polynomials, and `n` is the number of coefficients for each polynomial.

Cubic-spline interpolation:

Matlab Code

```
x = [7,9,11,12];
y = [49,57,71,75];
[breaks, coeffs, m, n] = unmkpp(spline(x,y))
```

In Command Window

```
breaks =
7 9 11 12
coeffs =
-0.3500 2.8500 -0.3000 49.0000
-0.3500 0.7500 6.900 57.0000
-0.3500 -1.3500 5.7000 71.0000
m =
3
n =
4
```

Programming Errors and Best Practices

(How can we make a good program?)

Steps for developing a computer solution:

1. State the problem concisely.

2. Specify the data to be used by the program. This is the "input."

3. Specify the information to be generated by the program. This is the "output."

4. Work through the solution steps by hand or with a calculator; use a simpler set of data if necessary.

Steps for developing a computer solution (continued)

5. Write and run the program.

6. Check the output of the program with your hand solution.

7. Run the program with your input data and perform a reality check on the output.

8. If you will use the program as a general tool in the future, test it by running it for a range of reasonable data values; perform a reality check on the results.

Effective documentation can be accomplished with the use of

1. Proper selection of variable names to reflect the quantities they represent.

2. Use of comments within the program.

3. Use of structure charts.

4. Use of flowcharts.

5. A verbal description of the program, often in *pseudocode*.

Finding Bugs

Debugging a program is the process of finding and removing the "bugs," or errors, in a program. Such errors usually fall into one of the following categories.

1. Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.

2. Errors due to an incorrect mathematical procedure. These are called *runtime errors*. They do not necessarily occur every time the program is executed; their occurrence often depends on the particular input data. A common example is division by zero.

To locate a runtime error, try the following:

1. Always test your program with a simple version of the problem, whose answers can be checked by hand calculations.

2. Display any intermediate calculations by removing semicolons at the end of statements.

3. To test user-defined functions, try commenting out the `function` line and running the file as a script.

4. Use the debugging features of the Editor/Debugger, which is discussed before.

Debugging Script Files

Program errors usually fall into one of the following categories.

1. Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.
2. Errors due to an incorrect mathematical procedure, called *runtime errors*. Their occurrence often depends on the particular input data. A common example is division by zero.

To locate program errors, try the following:

1. Test your program with a simple version of the problem which can be checked by hand.
2. Display any intermediate calculations by removing semicolons at the end of statements.
3. Use the debugging features of the Editor/Debugger.

Programming Style

1. *Comments section*

a. The name of the program and any key words in the first line.

b. The date created, and the creators' names in the second line.

c. The definitions of the variable names for every input and output variable. Include definitions of variables used in the calculations and *units of measurement for all input and all output variables!*

d. The name of every user-defined function called by the program.

Programming Style (continued)

2. *Input section* Include input data and/or the input functions and comments for documentation.

3. *Calculation section*

4. *Output section* This section might contain functions for displaying the output on the screen.

Example of a Script File

Problem:

Plotting the response of a single degree -second order differential equation

Example of a Script File (continued)

```
1       % Example 1
2 -     clear all; clf; clc
3
4       %Inputs
5 -     m=5; % mass
6 -     c=0; %damping
7 -     k=16; %stiffness
```

Example of a Script File (continued)

```
 8
 9 -    Tf=10; %Final time, sec
10 -    dt=1e-3; %time Step
11
12      %initial conditions
13 -    xo=0.01; % initial displacement
14 -    vo=3; % initial velocity
```

Example of a Script File (continued)

```
16 -    no_data_points=Tf/dt; %number of data points to plot
17
18 -    C= c^2-4*m*k;
19
20      % Overdamped System C^2>4mk
21 -    if C>0
22 -    p1= (-c+sqrt(c^2-4*m*k))/(2*m);
23 -    p2= (-c-sqrt(c^2-4*m*k))/(2*m);
24 -    A1=(xo*p2-vo)/(p2-p1);
25 -    A2=(vo-xo*p1)/(p2-p1);
26 -    for i=1:no_data_points+1
27 -    t(i)=(i-1)*dt;
28 -    x(i)=A1*exp(p1*t(i))+A2*exp(p2*t(i));
29 -    end
30 -    end
```

Example of a Script File (continued)

```
32      % Critically damped System C^2=4mk
33 -    if C==0
34 -    p1_1= (-c/(2*m));
35 -    c1=xo;
36 -    c2=vo-xo*p1_1;
37 -    for i=1:no_data_points+1
38 -    t(i)=(i-1)*dt;
39 -    x(i)=(c1+c2*t(i))*exp(p1_1*t(i));
40 -    end
41 -    end
```

Example of a Script File (continued)

```
43      % Undrdamped System C^2<4mk
44 -    if C<0
45 -    alpha= (-c)/(2*m);
46 -    beta= (sqrt(4*m*k-c^2))/(2*m);
47 -    X=sqrt(xo^2+((vo-alpha*xo)/beta)^2);
48 -    phi=atand((beta*xo)/(vo-alpha*xo));
49 -    for i=1:no_data_points+1
50 -    t(i)=(i-1)*dt;
51 -    x(i)=X*exp(alpha*t(i))*sin(beta*t(i)+phi*pi/180);
52 -    end
53 -    end
54
```

Example of a Script File (continued)

```
54
55      %plotting
56 -    figure(1);plot(t,x,'linewidth', 2);
57 -    xlabel('Time (sec)','FontSize',12);
58 -    ylabel('Displacement (mm)','FontSize',12);
59 -    axis([0 Tf+0.2 1.2*min(x) 1.2*max(x)])
60 -    title('Example 1.3 ','FontSize',16);
61 -    grid on
62 -    saveas(gcf, 'Example 1_3.tiff');
```