Chapter 1: MATLAB Basics

We will give a crash introduction of MATLAB in general before we start our learning journey on programming finite elements in MATLAB. The MATLAB software package is used for computation in engineering, science, and applied mathematics. It offers a powerful *interactive* programming language, excellent graphics, and a large standard library. In our crash introduction, we will cover five topics from MATLAB: **Basics** (this Chapter), **Matrices**, **Graphics**, **Scripts** (M-file), **Functions**, and **Miscellany**. Understanding of these topics will get you started on learning the wonderland of MATLAB and programming finite elements in MATLAB. See MATLAB on-line documents or the books by Higham, D. and Higham, N. (*MATLAB Guide, Second Edition*, SIAM, 2005) and Palm III, W. J. (*Introduction to MATLAB for Engineers, 3rd Edition*) if you would like to go deeper in MATLAB.

1.1 Getting Started with MATLAB

When you start MATLAB, you get a multipaneled desktop, as seen in Figure 1. The layout and behavior of the desktop and its components are highly customizable. The component that is the heart of MATLAB is called the **Command Window**, located in the middle by default. Here you can give MATLAB commands typed at the prompt, shown as **>>**. Unlike C++, Java, Fortran and other compiled computer languages, MATLAB is an interpreted environment—you give a command, and MATLAB tries to execute it right away, then awaits another.

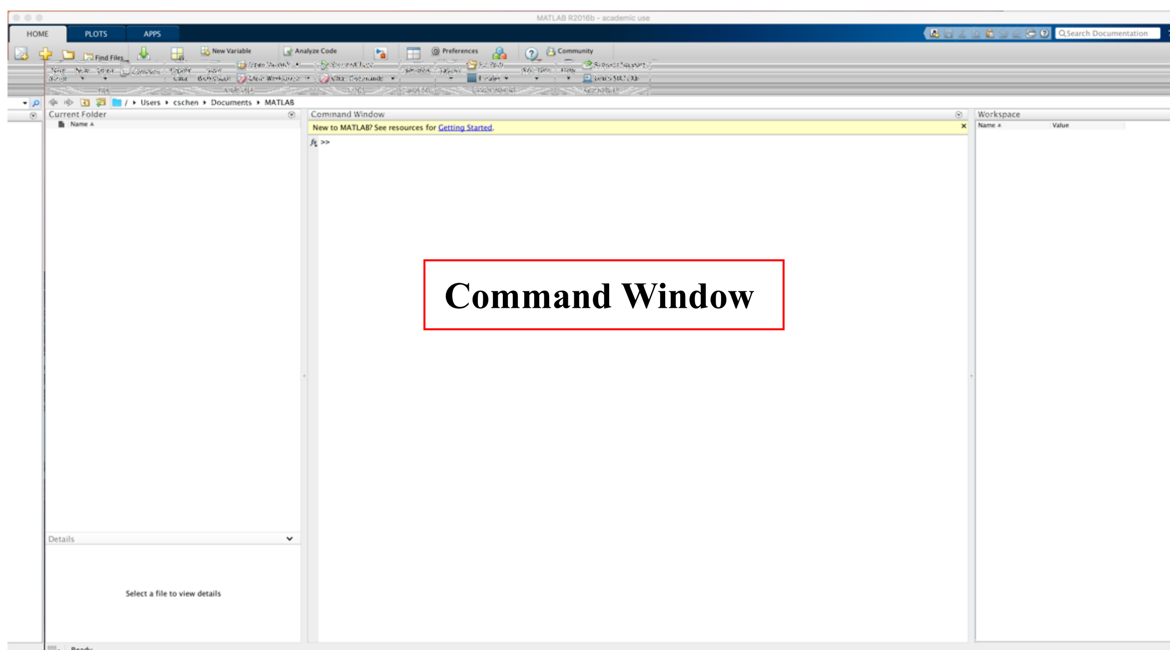


Figure 1.1 My initial MATLAB desktop window (R2016b).

(Demo)

```
>> a=[1 2 3]
a =
     1     2     3
```

(From A to A+) The best way to learn MATLAB is by trying it yourself, and you should experiment as you proceed, keeping the following points in mind.

- Upper and lower case characters are not equivalent (MATLAB is case sensitive and `a` and `A` are distinct variables.)
- Typing the name of a variable will cause MATLAB to display its current value.
- A semicolon at the end of a command suppresses the screen output.
- MATLAB uses parentheses, (), square brackets, [], and curly braces, {}, and these are not interchangeable. We will talk more on these topics in the sequel.
- The up arrow and down arrow keys can be used to scroll through your previous commands.

1.2 Help

MATLAB is huge so it is essential that you become familiar with the online help. There are two levels of help:

1. If you need quick help on the syntax of a command, type help in the command window. For example,

   ```
   >> help plot
   ```

   shows directly in the Command Window all the ways in which you can use the plot command. Typing help by itself gives you a list of categories that themselves yield lists of commands.

2. Typing doc followed by a command name brings up more extensive help in a separate window. For example, doc plot is better formatted and more informative than help plot. In the left panel one sees a hierarchical, browsable display of all the online documentation.

3. If you have some vague ideas but do not know the exact name of MATLAB command, you can try Help Navigator from Product Help. For example, if you type `Bessel`, you will be able to find the MATLAB functions for different kinds of Bessel functions. You can also ask around or google ☺

1.3 Things about MATLAB that are very nice to know

Below are a few features of MATLAB that are nice to know but often do not come to the attention of beginners.

- If a computation is taking too long, interrupt it by pressing **Ctrl-C** (after making sure the Command Window is active in the operating system).
- MATLAB has several predefined special constants and below is a short list:

Command	Description
`ans`	Temporary variable containing the most recent answer.
`eps`	Specifies the accuracy of floating point precision.
`Inf`	Infinity
`Nan`	Indicates an undefined numerical result
`pi`	The number π

- MATLAB by default displays only 4–5 digits of a result, but it actually stores about 15 significant digits. (You can see them all by typing format long.)

```
>> a=[1 2 3]
a =
     1     2     3
>> sqrt(a)
ans =
    1.0000    1.4142    1.7321
>> format long
>> ans
ans =
   1.000000000000000   1.414213562373095   1.732050807568877
>> format
>> ans
ans =
    1.0000    1.4142    1.7321
```

The `format` command controls how numbers appear on the screen. Table 1.1 gives the variants of this command.

Table 1.1: Numeric Display Formats in MATLAB

Command	Description and example
`format short`	Four decimal digits (the default); 13.6745.
`format long`	16 digits; 17.27484029463547.
`format short e`	Five digits (four decimals) plus exponent; 6.3792e+03.
`format long e`	16 digits (15 decimals) plus exponent; 6.379243784781294e−04.
`format bank`	Two decimal digits; 126.73.
`format +`	Positive, negative, or zero; +.
`format rat`	Rational approximation; 43/7.
`format compact`	Suppresses some blank lines.
`format loose`	Resets to less compact display mode.

2019/02/27

Chapter 2: MATLAB Matrices

The heart and soul of the MATLAB software is *linear algebra*. In fact, “MATLAB” was originally a contraction of “matrix laboratory.” More so than any other language, MATLAB **encourages and expects** you to make heavy use of matrices. Matrices are particularly important when we start programming finite elements in MATLAB.

An **m-by-n** matrix is a two-dimensional array of numbers consisting of **m rows** and **n columns**. Special cases are a column vector (n = 1) and a row vector (m = 1).

2.1 Matrix Generation

The most straightforward way to construct a small matrix is by enclosing its elements in **square brackets**. Use **spaces or commas to separate columns**, and use **semicolons or new lines to separate rows**:

```
>> A =  1 2 3; 4 5 6; 7 8 9
A =
     1     2     3
     4     5     6
     7     8     9

>> b=[0;1;0]
b =
     0
     1
     0
```

Information about the size is stored in matrices

```
>> si e(A)
ans =
     3     3
>> si e(b)
ans =
     3     1
```

Matrices can be built out of other matrices, **as long as the sizes are compatible**

```
>> [A b]
ans =
     1     2     3     0
     4     5     6     1
     7     8     9     0
```

Q: Can we do

```
>> [A; b]
```

A:

No since all rows in the bracket must have same number of columns and MATLAB will give you a helpful response interactively and instantly.

```
>> [A;b]
??? Error  sing ==>  ertcat
CAT arg ments dimensions are not consistent.
```

The transpose operation ( ) interchanges the rows and columns.

```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>>  '

ans =

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

We can also generate matrices by MATLAB pre-defined functions, such as random matrices which are quite useful for scientific computing in general:

```
>> rand(3)
ans =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

returns a 3-by-3 matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval **(0, 1)**. Similarly, `rand(M,N)` returns an M-by-N matrix.

Example: let us generate <u>10</u> rolls of one die.

Ans:

Other useful elementary matrix generation includes eros, ones, e e:

```
>>  eros(3,4)
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
>> ones(2)
ans =
     1     1
     1     1
>> e e(2,3)
ans =
     1     0     0
     0     1     0
```

e e is for an identify matrix.

Q: Use e e and si e to set up an identify matrix whose dimensions match those of a given matrix A

A:

2.2 Colon Notation and Referencing Elements

To enable access and assignment to submatrices, MATLAB has powerful notation based on the **colon** character. The colon is used to define vectors that can act as **subscripts**. For integers i and j, i:j denotes the row vector of integers from i to j (in steps of 1). A nonunit step s is specified as i:s:j. This notation is valid even i:9 ( ) -30 (us) 91ti:j

```
     4     3     2     1     0    -1    -2
>> 0:.75:3
ans =
     0    0.7500    1.5000    2.2500    3.0000
```

Single elements of a matrix are accessed as `A(i,j)`, where `i > 1` and `j > 1` (zero or negative subscripts are not supported in MATLAB). The submatrix comprising the intersection of rows `p` to `q` and columns `r` to `s` is denoted by `A(p:q, r:s)`. As a special case, a lone colon as the row or column specifier covers all entries in that row or column; thus `A(:,j)` is the jth column of `A` and `A(i,:)` the ith row. The keyword `end` used in this context denotes the last index in the specified dimension; thus `A(end,:)` picks out the last row of `A`. In effect, a lone colon is short for `1:end`.

An arbitrary **submatrix** can be selected by specifying the individual row and column indices. For example, `A([i j k],[p q])` produces the submatrix given by the intersection of rows `i`, `j`, and `k` and columns `p` and `q`.

```
>> B = [2,4,10,13;16,3,7,18;8,4,9,25;3,12,15,17];
>> C = B(2:3,1:3);
>> B

B =

     2     4    10    13
    16     3     7    18
     8     4     9    25
     3    12    15    17

>> C

C =

    16     3     7
     8     4     9
```

More example and questions:

```
>> A=[2 3 5; 7 11 13; 17 19 23]
A =
     2     3     5
     7    11    13
    17    19    23
```

Q:

```
>> A(2,1)
```

A:

Q:
```
>> A(2:3,2:3)
```
A:

Q:
```
>> A(:,1)
```
A:

Q:
```
>> A(2,:)
```
A:

Q:
```
>> A(end:-1:1, end)
```
A:

Q:
```
>> A([1 3],[2 3])
```
A:

Related to the colon notation for generating vectors of equally spaced numbers is the function `linspace` (linearly space vector). `linspace(a,b,n)` generates `n` equally spaced points between `a` and `b`. If `n` is omitted it defaults to 100. Example:

```
>> linspace(-1,1,9)
ans =
  Col mns 1 thro gh 8
   -1.0000  -0.7500  -0.5000  -0.2500   0  0.2500  0.5000  0.7500
  Col mn 9
    1.0000
```

The notation `[]` denotes an empty, 0-by-0 matrix. Assigning `[]` to a row or column is one

way to delete that row or column from a matrix:

```
>> A(2,:) = []
A =
     2     3     5
    17    19    23
```

Q: How can you achieve the same thing (erasing the second row) using colon notation?
A:

2.3 Matrix and Array Operations

MATLAB operations can be carried out in a matrix sense (**according to the rules of matrix algebra**) or an array sense (**elementwise**). Table 2.1 summarizes the syntax.

Table 2.1 Elementary matrix and array operations in MATLAB

Operation	Matrix sense	Array sense

(From A to A+) MATLAB has defined a right division as well as a left division operator. Their mathematical equivalency is given below:

MATLAB notation	Mathematical equivalent
Right division: `a/b`	$\frac{a}{b}$
Left division: `a\b`	$\frac{b}{a}$

The matrix operations follow the rule you have learned from matrix algebra: Addition and subtraction are defined for matrices of the same dimension. The product `A*B` is the result of matrix multiplication, defined only when the number of columns of `A` and the number of rows of `B` are the same. Examples:

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
```

```
>> B=ones(2)
B =
     1     1
     1     1
>> A+B
ans =
     2     3
     4     5
>> A*B
ans =
     3     3
     7     7
>> A\B
ans =
    -1    -1
     1     1
>> A*ans
ans =
     1     1
     1     1
```

Multiplication and division in the array sense (**elementwise sense**) are specified by preceding the operator **with a period**. If A and B are matrices of the same dimensions then `C = A.*B` sets `C(i,j)=A(i,j)*B(i,j)` and `C = A./B` sets `C(i,j)=A(i,j)/B(i,j)`. The assignment `C =A.\B` is equivalent to `C = B./A`. With the same A and B as in the previous example:

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
>> B=ones(2)
B =
     1     1
     1     1
>> A.*B
ans =
     1     2
     3     4
>> A.\B
ans =
    1.0000    0.5000
    0.3333    0.2500
>> A.*ans
ans =
     1     1
     1     1
```

As we have mentioned, the transpose of the matrix `A` is obtained with `A'`. For the special case of column vectors and , `'*` is the inner, scalar, or dot product, which can also be obtained using the dot function as `dot( , )`. The vector or cross product of two 3-by-l or 1-by-3

vectors (as used in mechanics) is produced by `cross`. Example:

```
>>  =[-1 0 1]'; =[3 4 5]';
>>  '*
ans =
     2
>> dot( , )
ans =
     2
>> cross( , )
ans =
    -4
     8
    -4
```

We use left division regularly to solve the well-known linear algebra problems **Ax = b**

```
>>  =A\B
```

<u>Example</u>: Consider a system of three linear equations with three unknowns (x_1, x_2, x_3):

$$
\begin{aligned}
3x_1 + 2x_2 - x_3 &= 10 \\
-x_1 + 3x_2 + 2x_3 &= 5 \\
x_1 - x_2 - x_3 &= -1
\end{aligned}
$$

The system of equations can be rewritten in matrix form as:

$$
\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}
$$

In MATLAB:

```
>> A = [3 2 -1; -1 3 2; 1 -1 -1]

A =

     3     2    -1
    -1     3     2
     1    -1    -1

>> b = [10; 5; -1]
```

```
b =

    10
     5
    -1

>>    = A\b

  =

   -2.0000
    5.0000
   -6.0000

>> check = A*

check =

   10.0000
    5.0000
   -1.0000
```

2019/02/27

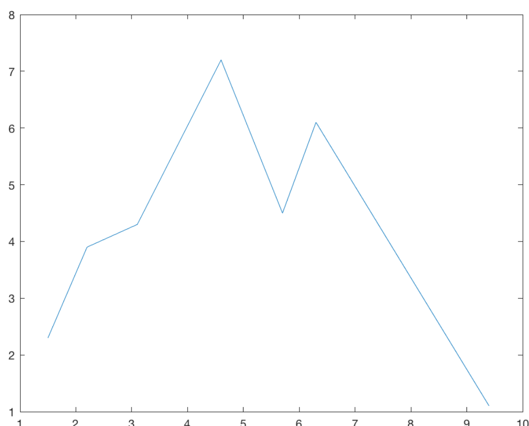
Chapter 3: Graphics

Graphical display is one of the greatest strengths of the MATLAB and we will cover some basic ones in this short tutorial. You can find many more details and options in the online documentation.

Ultimately, any plot comes down to the display of numerical values. Broadly speaking, the source of data can be either generated **(1) by explicit mathematical functions** or **(2) by observed data or discrete values of an unknown function**. MATLAB has a number of functions, all starting with the letters `e` , for plots made by explicit mathematical functions. The `e plot` will be covered later when we introduce function handles later. The "traditional" and better-known plotting functions `plot` in MATLAB are used to plot discrete values and will be covered in this Chapter.

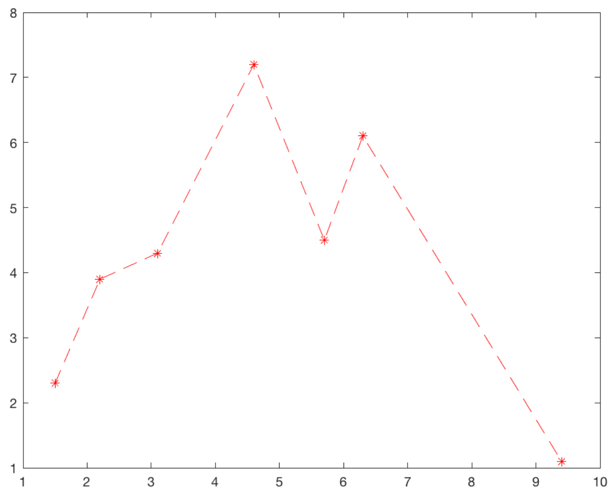
For discrete data in two dimensions, we use `plot` to "connect the dots".

```
>>   =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>>   =[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
>> plot( , )
```



More generally, we could replace `plot( , )` with `plot( , ,        )`, where string combines up to **three elements** that control the **color**, **marker**, and **line style**. For example, `plot( , ,'r*--')` specifies that a red asterisk is to be placed at each point x(i), y(i) and that the points are to be joined by a red dashed line whereas `plot( , ,'k+')` specifies that a black cross marker with no line joining the points.

```
>> plot( , ,'r*--')
```




```
>> plot( , ,'k+')
```

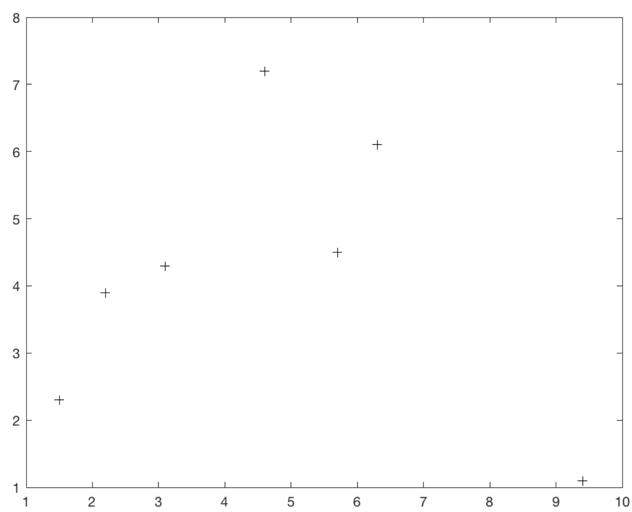


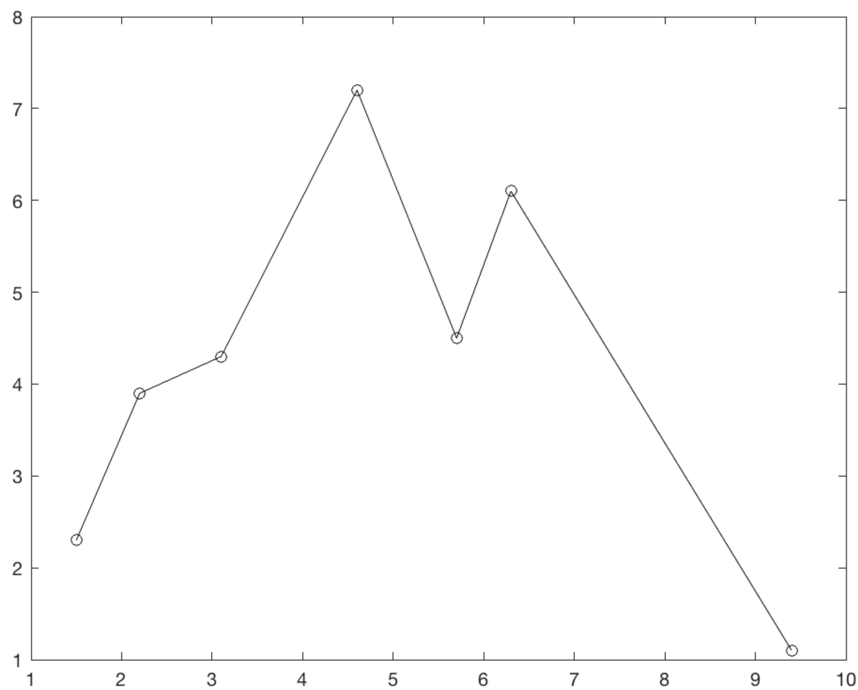
Table below lists the options available in MATLAB

Color	
`r`	Red
`g`	Green
`b`	Blue

Marker	
`o`	Circle
`*`	Asterisk
`.`	Point
`+`	Plus
`x`	Cross

Line style	
`-`	Solid line (default)

Q: Given the same data, what is the corresponding `plot` command to generate the following figure?



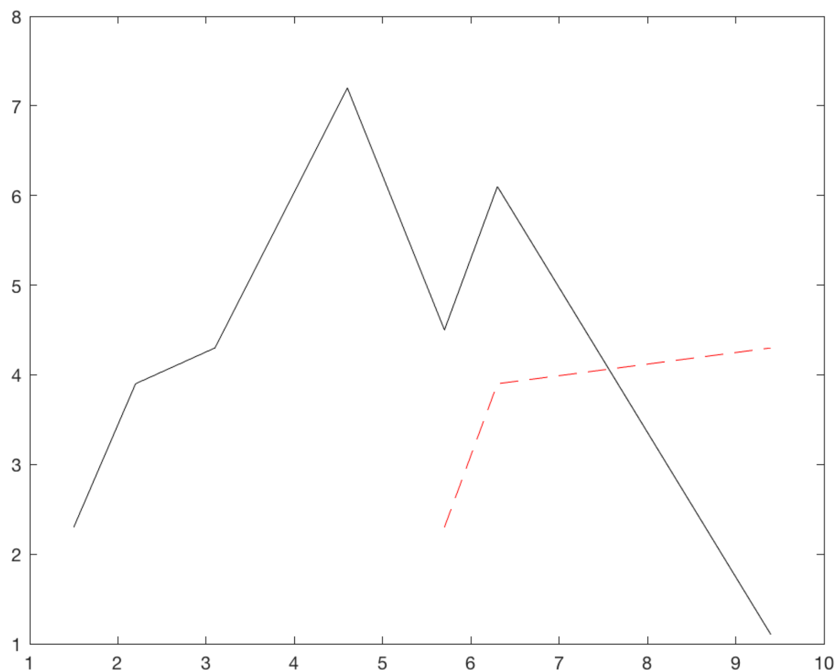
A:

More than one set of data can be passed to plot. For example,

```
>>  =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>>  =[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
```

```
>> b= (5:7);
>> c= (1:3);
>>      (    '   '       '    ')
```

superimposes plots of x(i), y(i) and b(i), c(i) with solid black and dashed red line styles, respectively.



You can achieve the same thing (a figure with multiple sets of data) via `hold on` command (the default is `hold off`).

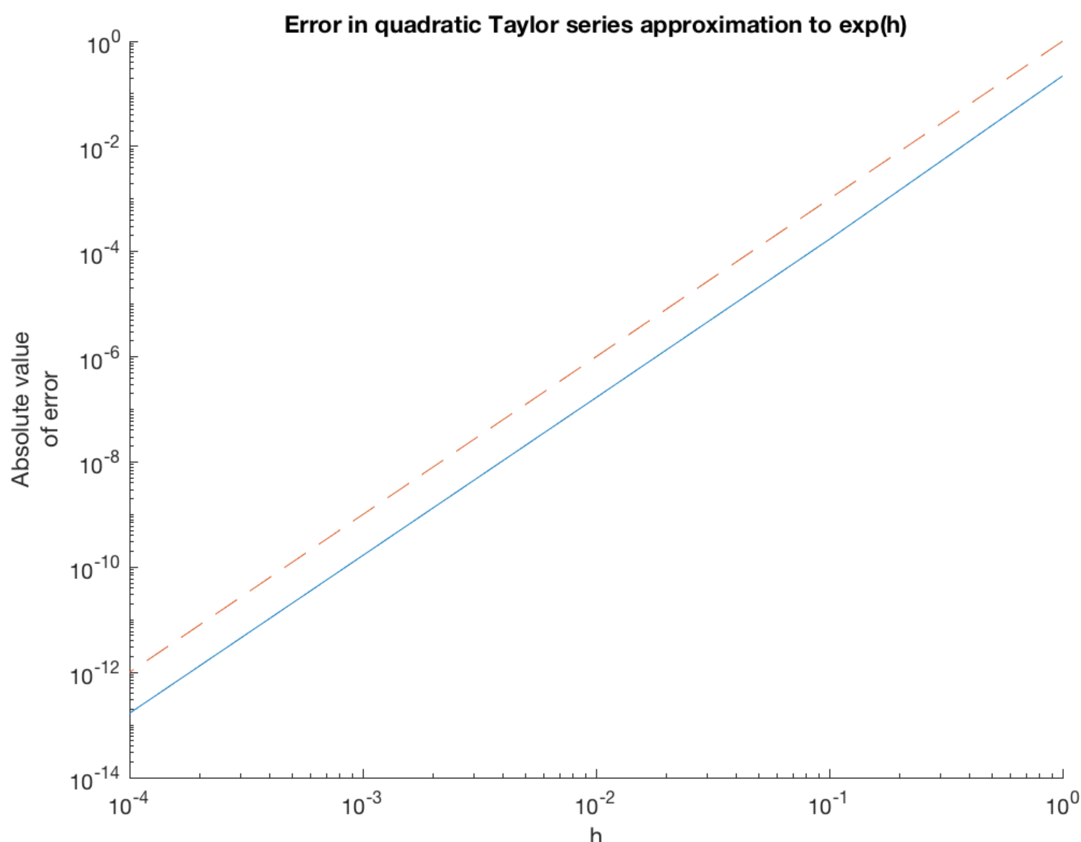
```
>>   =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>>   =[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
>>      (    '   ')
>> b= (5:7);
>> c= (1:3);

>>      (    '    ')
```

(From A to A+) If one plotting command is later followed by another then the new picture will either replace or be superimposed on the old picture, depending on the current **hold** state. Typing `hold on` causes subsequent plots to be superimposed on the current one, whereas `hold off` specifies that each new plot should start from fresh. The default status corresponds to `hold off`.

Using `loglog` instead of `plot` causes the axes to be scaled **logarithmically**. This feature is

useful for revealing **power-law relationships** as straight lines. In the example below we plot $\left|(1+h+\frac{h^2}{2})-\exp(h)\right|$ against h for $h=1,\ 10^{-1},\ 10^{-2},\ 10^{-3},\ 10^{-4}$. This quantity behaves like a multiple of h^3 when h is small, and hence on a log-log scale the values should lie close to a straight line of slope 3. To confirm this, we also plot a dashed reference line with the predicted slope, exploiting the fact that more than one set of data can be passed to the plot commands.

```
>> h = 10.^[0:-1:-4];
>> ta lerr = abs((1+h+h.^2/2) - e p(h));
>> loglog(h,ta lerr,'-',h,h.^3,'--')
>>  label ('h')
>>  label( 'Absol te  al e','of error' )
>> title('Error in q adratic Ta lor series appro imation to
e p(h)')
>> bo  off
```



In this example, we used `title`, ` label`, and ` label`. These functions reproduce their input string above the plot and on the x- and y-axes, respectively. The multiline y-axis label is created by a **cell array** of strings, one for each line (see Chapter 5). We also used the command `bo  off`, which removes the box from the current plot, leaving just the x- and y-axes.

Related functions are `semilog` and `semilog`, for which only the x- or y-axis, respectively, is logarithmically scaled.

2019/02/27

Chapter 4: More MATLAB Basic Elements

In this Chapter, we will introduce more basic elements of MATLAB that are not covered in the previous Chapters. By the end of the Chapter, you will be able to write simple but functional MATLAB programs.

4.1 Script

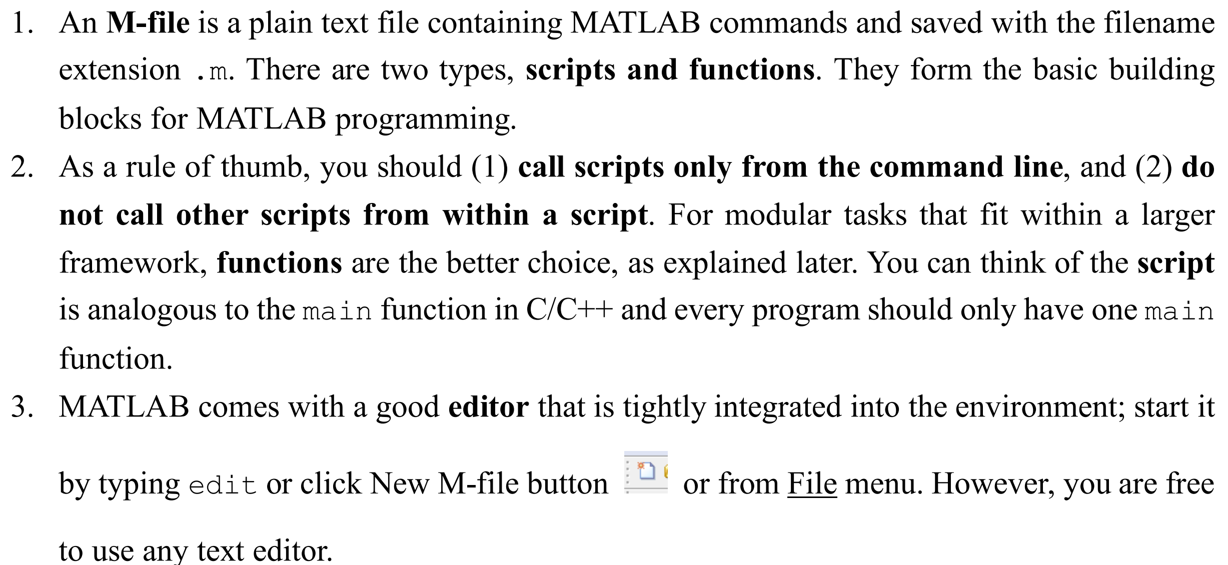
Instead of typing commands directly in the Command Window, a series of commands can be placed into a file, and the entire file can be executed by typing its name in the Command Window. Such files are called **script files**. Script files (and functions, which we will see later) are also known as **M-files**, because they have a file extension of ".m".

The contents of a script file **are interpreted as though they were typed at the command prompt**. To perform a sequence of related commands, you can write them into a script. For example, suppose you wish to process a set of exam marks using the MATLAB functions `sort`, `mean`, `median`, and `std`, which, respectively, sort into order and compute the arithmetic mean, the median, and the standard deviation. You can create a script file, say `e amMarks.m` of the form

```
% e am mark
e mark = [12 0 5 28 87 3 56];
e sort = sort(e mark)
e mean = mean(e mark)
e med = median(e mark)
e std = std(e mark)
```

Typing `e amMarks` at the command line then produces the output

```
>> e amMark
e sort =

     0     3     5    12    28    56    87
e mean =

   27.2857
e med =

    12
e std =

   32.8010
```



Remarks:

1. An **M-file** is a plain text file containing MATLAB commands and saved with the filename extension `.m`. There are two types, **scripts and functions**. They form the basic building blocks for MATLAB programming.
2. As a rule of thumb, you should (1) **call scripts only from the command line**, and (2) **do not call other scripts from within a script**. For modular tasks that fit within a larger framework, **functions** are the better choice, as explained later. You can think of the **script** is analogous to the `main` function in C/C++ and every program should only have one `main` function.
3. MATLAB comes with a good **editor** that is tightly integrated into the environment; start it by typing `edit` or click New M-file button or from File menu. However, you are free to use any text editor.

Comment

An important type of statement in any M-file is a comment, which is indicated by a percent sign `%`. Any text on the same line after a percent sign is ignored. Furthermore, the first contiguous block of comments in an M-file serves as documentation for the file and will be typed out in the command window if help is used on the file. For instance, say the following is saved as `m script.m` on the path:

```
% This script is o r first MATLAB script
% for demo.
  = rand(1); % a random n mber
```

Then at the prompt one would could obtain

```
>> help m script
  This script is o r first MATLAB script
  for demo.
```

4.1.1 Initializing Variables with Keyboard Input

It is possible to prompt a user and initialize a variable with data that the user types directly at the keyboard. This option allows a script file to prompt a user for input data values while it is executing. The input function displays a prompt string in the Command Window and then waits for the user to type in a response. For example, consider the following statement:

```
m _ al = inp t('Enter an inp t  al e:');
```

When this statement is executed, MATLAB prints out the string `'Enter an inp t  al e:'`, and then waits for the user to respond. If the user enters a single number, it may be simply typed in. If the user enters an array, it must be enclosed in brackets. Below are two examples (user inputs are shown in boldface):

```
>> m _ al = inp t('Enter an inp t  al e:');
Enter an inp t  al e:
>> m _ al

m _ al =

    3.4000
```

```
>> m _ al = inp t('Enter an inp t  al e:');
Enter an inp t  al e:
>> m _ al

m _ al =
```

```
>> str = ['The  al e of pi = ' n m2str(pi)];
>> disp(str)
The  al e of pi = 3.1416
```

The first statement creates a string array containing the message, and the second statement displays the message.

4.2.2 Formatted output with the `fprintf` function

An even more flexible way to display data is with the `fprintf` function. The `fprintf` function displays one or more values together with related text and lets the programmer control the way in which the displayed value appears.

The general form of `fprintf` function when it is used to print to the Command Window is

```
fprintf(format,data)
```

where format is a string describing the way the data is to be printed, and data is one or more scalars or arrays to be printed. The format is a character string containing text to be printed along with special characters describing the format of the data. For example, the function

```
fprintf('The  al e of pi is %f \n',pi)
```

will print out `'The  al e of pi is 3.141593'` followed by a line feed. The characters `%f` are called conversion characters; they indicate that the a value in the data list should be printed out in floating-point format at that location in the format string. The characters `\n` are escape characters; they indicate that a line feed should be issued so that the following text starts on a new line. There are many types of conversion characters and escape characters that may be used in an `fprintf` function. A few common ones are listed below:

Format String	Results
%d	Display value as an integer.
%e	Display value in exponential format.
%f	Display value in floating-point format.
%g	Display value in either floating-point or exponential format, whichever is shorter.
\n	Skip to a new line.

It is also possible to specify the width of the field in which a number will be displayed and the number of decimal places to display. This is done by specifying the width and precision after

the % sign and before the f. For example, the function

```
fprintf('The  al e of pi is %6.2f \n',pi)
```

will print out 'The al e of pi is 3.14' followed by a line feed. The conversion characters %6.2f indicate that the first data item in the function should be printed out in floating-point format in a field six characters wide, including two digits after the decimal point.

Example —Temperature Conversion

Write a MATLAB script that reads an input temperature in degrees Fahrenheit, converts it to an absolute temperature in kelvin, and writes out the result. We know that

$$T(\text{in kelvin}) = \left[\frac{5}{9}T(\text{in °F}) - 32.0\right] + 273.15$$

A sample run is given below:

```
>> temp_con ersion
Enter the temperat re in degrees Fahrenheit: 75
 75.00 degrees Fahrenheit = 297.04 kel in.
```

Ans

Chapter 5: Branches and Loops

In this Chapter, we will introduce a number of MATLAB statements that allow us to **control** the order in which statements are executed in a program. There are two broad categories of control statements: (1) **branches**, which select specific sections of the code to execute and (2) **loops**, which cause specific sections of the code to be repeated.

With the introduction of branches and loops, our programs are going to become more complex, and it will get easier to make mistakes. You should **ALWAYS** program with caution and **DO NOT** write your codes if you have not figured out what to write.

5.1 Branches: `if` and `s itch`

Often, a program needs to branch based on run-time conditions. MATLAB offers features for this that are similar to those in most languages.

5.1.1 ____________________ for a small number of alternatives

The general form of the `if` statement is

```
if logical e pression 1
    statement gro p 1
         logical e pression 2
    statement gro p 2
else
    statement gro p 3
```

The `else` and `elseif` statements may be omitted if not required. However, if both are used, the `else` statement must come after the `elseif` statement to take care of all conditions that might be unaccounted for.

<u>Example</u>

```
  =-1
if  >=0
    disp('Absol te  al e is: ')
     =
elseif   < 0
    disp('Absol te  al e is: ')
     =-
else
```

```
    disp('Something  rong');
end
```

Q: what are the outputs?

A:

It is possible to write a complete `if` on a single line by separating the parts of the construct by commas or semicolons. Thus the following two constructs are identical:

```
        < 0
        = abs( );
```

and

```
        < 0;    = abs( );
```

```
        < 0,    = abs( );
```

However, this should be done only for very simple constructs.

5.1.2 _______ for a large number of alternatives

The switch construct is another form of branching construct. It permits a programmer to select a particular code block to execute based on the value of a single integer, character, or logical expression. The general form of a `s itch` construct is

```
s itch (s itch_e pr)
case case_e pr_1
  Statement 1
  Statement 2
  ...
case case_e pr_2
  Statement 1
  Statement 2
  ...
...
other ise
  Statement 1
  Statement 2
  ...
end
```

The switch expression can be a **string** or a **number**. The first matching case has its commands executed. C programmers should note that MATLAB's `s itch` construct behaves differently from that in C: once a MATLAB `case` group expression has been matched and its statements executed, control is passed to the first statement after the switch, with no need for `break` statements.

Example

```
 nits = 'time';
s itch  nits
   case 'length'
      disp('meters')
   case ' ol me'
      disp('liters')
   case 'time'
      disp('seconds')
   other ise
      disp('I gi e  p')
end
```

Q: what are the outputs?

A:

5.2 Loops: for and while

We often need to run a repetitive execution of a block of statements. Again, MATLAB is similar to other languages.

5.2.1 ___ loop for a fixed number of iterations

The `for` loop is a loop that executes a block of statements a specified number of times. The `for` loop has the form:

```
for inde  = e pr
  ...
  ...
  ...
end
```

Let us consider an example for computing 10 members of the famous Fibonacci sequence:

```
>> f = [1 1];
for n =
   f(n) = f(n-1) + f(n-2);
```

```
end

>> f
f =
     1     1     2     3     5     8    13    21    34    55
```

You can have as many statements as you like in the body of a loop. In this example, the value of the index `n` will change from 3 to 10, with an execution of the body after each assignment. Remember that the **colon** notation and `3:10` is a row vector.

Exercise: write a script (`forScript.m`) to compute the sum $\sum_{k=1}^{15} 5k^2 - 2k$ and report the results. Below is a sample run:

```
>> forScript
The s m for 15 terms is:
      5960
```

Ans:

5.2.2 ______ loop for repeating statements based on satisfying a condition rather than a fixed number of times

A `hile` loop is a block of statements that are repeated indefinitely as long as some condition is satisfied. The general form of a `hile` loop is

```
 hile e pression
  ...
  ...
  ...
end
```

<u>Example</u>

```
 hile abs( ) > 1
      =  /2;
end
```

The condition is evaluated before the body is executed, so it is possible to get zero iterations. It's often a good idea to limit the number of repetitions to avoid infinite loops (CTRL-C in case you run into an infinite loop).

```
n = 0;
 hile abs( ) > 1
     =  /2;
   n = n+1;

end
```

Exercise: write a script (`hileScript.m`) to determine the number of terms required for the sum of the series $5k^2 - 2k,\ k = 1, 2, 3, \ldots$ to exceed 10000. Report the number of terms and the sum. Below is a sample run:

```
>>  hileScript
The n mber of terms is:
   18

The s m is:
      10203
```

Ans:

5.3 Implied loop and vectorization to avoid loops

Many MATLAB commands contain *implied loops*. For example, consider these statements:

```
   = [0:5:100];
   = cos( )
```

To achieve the same result using a `for` loop, we must type

```
for k = 1:21
     = (k - 1)*5;
    (k) = cos( );
end
```

If you are familiar with C/C++, you might be inclined to solve problems in MATLAB using loops. To maximize the power of MATLAB, you might need to adopt a new approach to problem solving. As the preceding example shows, you often can save many lines of code by using MATLAB commands, instead of using loops. Your programs will also run faster because MATLAB was designed for high-speed vector computations.

Vectorization refers to the avoidance of for and while loops. As an example, suppose x is a **column vector** and you want to compute a matrix D such that $d_{ij} = x_i - x_j$. The standard C-like implementation would involve two nested loops:

```
n = length( ); % eq i alent to ma (si e( ))
D =  eros(n); % memor  preallocation
for j = 1:n
    for i = 1:n
        D(i,j) =  (i) -  (j);
    end
end
```

(From A to A+) We often do **memory preallocation** to save time (i.e., do all the required memory allocation at once) if we know the size of the matrix.

Back to our example, the loops could be written in either order here. But the innermost loop is easily replaced by a vector operation

```
n = length( ); % eq i alent to ma (si e( ))
D =  eros(n); % memor  preallocation
for j = 1:n
    D(:,j) =       (j)
end
```

MATLAB makes it easy to act on **vectors and matrices** as atomic objects (not on their individual entries as in C++ and FORTRAN). Thus, the style of vectorization is often preferable for MATLAB veterans.

Chapter 6: User-Defined Functions

Both functions and scripts gather MATLAB statements to perform complex tasks. The most important distinguishing feature of a function is its **local workspace**. The variables in the main program (script) are not visible to the function (except for those in the input argument list), and the variables in the main program cannot be accidentally modified by anything occurring in the function. Therefore, mistakes or changes in the function's variables cannot accidentally cause unintended side effects in the other parts of the program.

The restrictions on data access are called **scoping**, and they make it possible for us to write complex programs. When we program in MATLAB, it is wise to break large program tasks into functions whenever practical to achieve the important benefits of independent component testing, reusability, and isolation from undesired side effects.

6.1 Basics

Each function starts with a line such as

```
              o t1, o t2   = m f n( in1, in2, in3 )
```

The name `m f n` should match **the name of the file on disk** (`m f n.m`). The variables `in1` etc. are **input arguments**, and `o t1` etc. are **output arguments**. You can have as many input and output arguments as you like. In general, the **ONLY** communication between a function's workspace and that of its caller is through the input and output arguments.

The values of the input arguments to a function are **copies** of the original data, so any changes you make to them will not change anything outside of the function's scope.

(From A to A+) You can create a new function from a simple template from MATLAB File menu:

```
f nction [ o tp t_args ] = Untitled( inp t_args )
%UNTITLED S mmar  of this f nction goes here
%   Detailed e planation goes here


end
```

You can terminate any function with an `end` statement but, in most cases, this is **optional**. `end` statements are required only in M-files that employ one or more nested functions. Within such

an M-file, every function (including primary, nested, private, and subfunctions) must be terminated with an `end` statement. You can terminate any function type with `end`, but doing so is not required unless the M-file contains a nested function.

Example: (`q adform.m`) a function that implements the quadratic formula for finding the roots of $ax^2+bx+c=0$:

```
f nction [ 1, 2] = q adform (a,b,c)
%QUADFORM ret rns the roots of a ^2+b +c from
%   the q adratic form la
d = sqrt(b^2 - 4*a*c);
 1 = (-b + d) / (2*a);
 2 = (-b - d) / (2*a);
```

We can then **call** the function in the Command Window:

```
>> help q adform
 QUADFORM ret rns the roots of a ^2+b +c from
    the q adratic form la

>> [a b]=q adform(1,2,1)
a =
    -1
b =
    -1
>> [a b]=q adform(1,3,2)
a =
    -1
b =
    -2
>> [a b]=q adform(3,2,1)
a =
  -0.3333 + 0.4714i
b =
  -0.3333 - 0.4714i
```

Exercise: Write a function `circle` to compute the area and circumference of a circle, given its radius as an input argument.

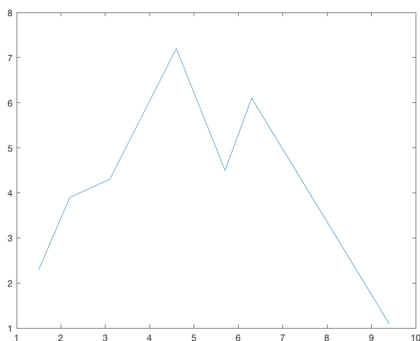
Ans:

(From A to A+) The main use of a function is to compartmentalize (劃分) a specific task. Any complex problem is decomposed into a series of smaller tasks, and the scoping rules of functions allow you to deal with each task independently. They also let you exploit well-crafted solutions to fundamental tasks that appear over and over in different problems. Like a good theorem, a good function invites you to inspect the details of its construction once and then **forget** about them.

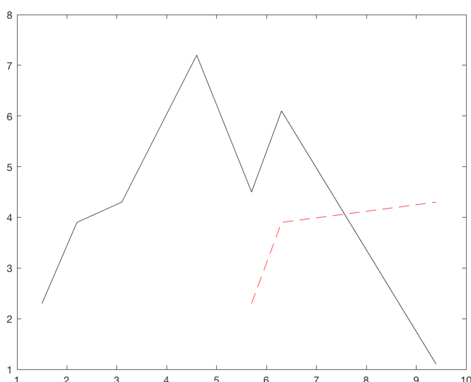
6.2 Optional Arguments

Many MATLAB functions support optional input arguments and output arguments. For example, we have seen calls to the `plot` function with as few as two or as many as six input arguments.

```
>>   =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>>   =[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
         (   )
```



```
>>   =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>>   =[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
>> b= (5:7);
>> c= (1:3);
>>       (    '  '      '   ')
```



How do MATLAB functions know how many input and output arguments are present, and how

do they adjust their behavior accordingly?

There are eight special functions that can be used by MATLAB functions to get information about their optional arguments, and to report errors in those arguments. Two of the commonly used functions are introduced here.

`nargin`—This function returns the number of actual input arguments that were used to call the function.
`nargo t`—This function returns the number of actual output arguments that were used to call the function.

When functions `nargin` and `nargo t` are called within a user-defined function, these functions return the number of actual input arguments and the number of actual output arguments that were used to when the user-defined function was called.

<u>Example</u>: Implement a home-made square root function `sqrtn` using a simple Newton iteration. Given $a > 0$, a simple Newton iteration to find $\sqrt{a}$ is:

$$x_{k+1} = \frac{1}{2}(x_k + \frac{a}{x_k}) \qquad x_0 = a$$

For example, $\sqrt{2}$ for the first four iterations and its relative change $\frac{|x_{k+1} - x_k|}{|x_{k+1}|}$ are:

```
k              _k                rel. change
 1:  1.5000000000000000e+000  3.33e-001
 2:  1.4166666666666665e+000  5.88e-002
 3:  1.4142156862745097e+000  1.73e-003
 4:  1.4142135623746899e+000  1.50e-006
```

Implement this home-made square root function with the following function prototype:

```
f nction [ ,iter] = sqrtn(a, tol)
```

in which `tol` is the tolerance for convergence $(\frac{|x_{k+1} \quad x_k|}{|x_{k+1}|} < \text{tol})$ (the default is `eps`) and `iter` is the number of iteration. Implement your function using `nargin` to determine whether you need to set the default tolerance.

Below are a few sample runs:

```
>> [  iter] = sqrtn(2, 0.01)
 k            _k          rel. change
 1:  1.5000000000000000e+00   3.33e-01
 2:  1.4166666666666665e+00   5.88e-02
 3:  1.4142156862745097e+00   1.73e-03

  =

    1.4142

iter =

     3

>> [  iter] = sqrtn(2)
 k            _k          rel. change
 1:  1.5000000000000000e+00   3.33e-01
 2:  1.4166666666666665e+00   5.88e-02
 3:  1.4142156862745097e+00   1.73e-03
 4:  1.4142135623746899e+00   1.50e-06
 5:  1.4142135623730949e+00   1.13e-12
 6:  1.4142135623730949e+00   0.00e+00

  =

    1.4142

iter =

     6
```

Ans: