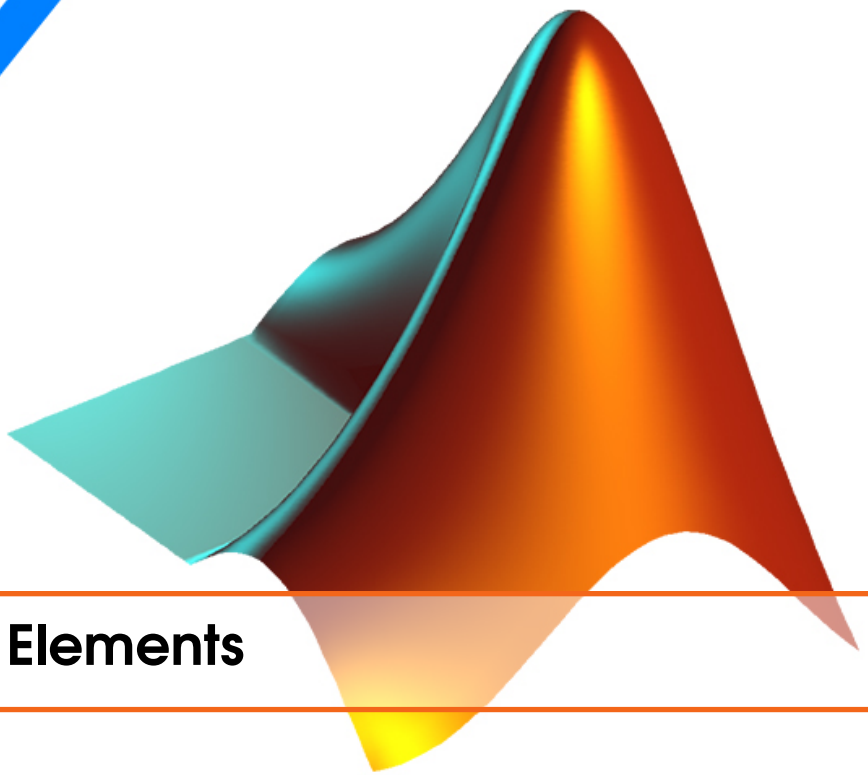
III Introduction to programming with Matlab

CYDOMEDIA

WHAT IS
MATLAB?
MATLAB PLOT AND
MATLAB FUNCTIONS



6. Programming Elements

So far, we have seen how to use MATLAB to execute commands or evaluate expressions directly from the command window (after the prompt »). These commands are typically entered as single statements, often written on a single line. However, some problems require multiple steps or conditional operations to describe their solutions properly for instance, determining the roots of a quadratic equation while considering all possible cases. A structured set of such instructions designed to solve a specific problem is called a program. In this section, we introduce the fundamental mechanisms for writing and executing programs in MATLAB, illustrating how complex computational tasks can be automated and organized effectively.

6.1 General:

6.1.1 Writing long expressions:

If a long expression cannot be written in a single line, it can be divided into several lines by putting at the end of each line at least three points.

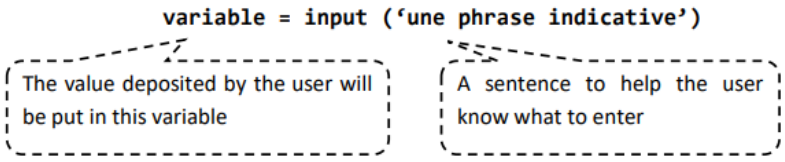
■ **Example 6.1** `>> (sin(pi/3)^2/cos(pi/3)^2)-(1-2*(5+sqrt(x)^5/(-2*x^3-x^2)^1+3*x)) ;`
This expression can be rewritten as follows:

```
>> (sin(pi/3)^2/cos(pi/3)^2)- ...
>> (1-2*(5+sqrt(x)^5 ...
>> /(-2*x^3-x^2)^1+3*x)) ;
```

■

6.1.2 Reading data in a program (Inputs):

To read a value given by the user, it is possible to use the input command, which has the following syntax:



When Matlab executes such an instruction, the indicative phrase will be displayed to the user waiting for the latter to enter a value.

■ **Example 6.2**
```
>> A = input ('Enter a whole number : ')
Enter a whole number : 5
A =
 5
>>
>> A = input (' Enter a whole number : ');
Enter a whole number : 5
>>
>> B = input ('Enter a vector line : ')
Enter a vector line : [1:2:8,3:-1:0]
B =
 1   3   5   7   3   2   1   0
```
■

6.1.3 Writing data in a program (Outputs):

We have already seen that MATLAB can display the value of a variable simply by typing its name at the command prompt.
For example:

```
>> A = 5 ;
>> A % Ask Matlab to display the value of A
A =
 5
```

With this method, Matlab writes the name of the variable (A) then the sign (=) followed by the desired value. However, there are cases where only the value of the variable is displayed (without the name and without the sign =). To do this, we can use the disp function, which has the following syntax:disp(object) The value of the object can be a number, a vector, a matrix, a string or an expression.
It is reported that with an empty vector or matrix, disp displays nothing.

■ **Example 6.3**
```
>> disp(A) % Display the value of A without 'A = '
 5
>> disp(A); % Semicolon has no effect
 5
>> B % Display vector B by the classical method
B =
 1  3  5  7  3  2  1  0
>> disp(B) % Display the vector B without 'B = '
 1  3  5  7  3  2  1  0
>> C = 3 :1 :0 % Creating an empty C vector
```

```
C =
 Empty matrix: 1-by-0
>> disp(C) % disp displays nothing if vector is empty
```

■

6.2 Logical expressions:

6.2.1 Logical operations:

Logical operations in MATLAB are used to compare values and return logical values (true/false or 1/0). These operations are essential for conditional statements, loops, and decision-making in your programs. Below are the common logical operations:

1. **Relational Operators** These compare two values and return `true (1)` or `false (0)`.
   - Equal to (==):
     ```
     result = (x == y); % Returns true if x is equal to y
     ```
   - Not equal to (=):
     ```
     result = (x ~= y); % Returns true if x is not equal to y
     ```
   - Less than (<):
     ```
     result = (x < y); % Returns true if x is less than y
     ```
   - Greater than (>):
     ```
     result = (x > y); % Returns true if x is greater than y
     ```
   - Less than or equal to (<=):
     ```
     result = (x <= y); % Returns true if x is less than or equal to y
     ```
   - Greater than or equal to (>=):
     ```
     result = (x >= y); % Returns true if x is greater than or equal to y.
     ```
2. **Logical Operators**
   These operators allow performing logical operations on arrays or scalar values.
   - **AND** (&):
     ```
     result = (x > 0) & (y > 0); % Returns true if both conditions are true
     ```
   - **OR** (|):
     ```
     result = (x > 0) | (y > 0); % Returns true if either condition is true
     ```
   - **NOT** (∽):
     ```
     result = ~(x > 0); % Returns true if x is not greater than 0
     ```
3. **Logical Arrays**
   In MATLAB, logical operations can be performed on arrays. The result is an array of logical values.

   ■ **Example 6.4**
   ```
   A = [1, 2, 3, 4];
   B = [4, 3, 2, 1];
   result = (A > B); % Compares element-wise, returns [false, false, true, true]
   ```
   ■

4. **Short-circuit Operators**
   These operators are used to perform logical operations but stop evaluating as soon as the result is determined.
   - Short-circuit AND (&&):

     ```
     result = (x > 0) && (y > 0); % Stops if the first condition is false
     ```

   - Short-circuit OR (||):

     ```
     result = (x > 0) || (y > 0); % Stops if the first condition is true
     ```

5. **Logical Functions**
   MATLAB also provides built-in logical functions for more complex operations:
   - **all():** Returns true if all elements are true.

     ```
     result = all(A > 0); % Returns true if all elements in A are greater than 0
     ```

   - **any():** Returns true if any element is true.

     ```
     result = any(A > 0); % Returns true if any element in A is greater than 0
     ```

   - **find():** Returns the indices of non-zero elements (true values).

     ```
     indices = find(A > 2); % Finds the indices where A is greater than 2
     ```

■ **Example 6.5**

```
>> x=10;
>> y=20;
>> x < y % displays 1 (true)
ans =
1
>> x <= 10 % displays 1 (true)
ans =
1
>> x == y % displays 0 (false)
ans =
0
>> (0 < x) & (y < 30) % displays 1 (true)
ans =
1
>> (x > 10) | (y > 100) % displays 0 (false)
ans =
0
>> ~(x > 10) % displays 1 (true)
ans =
1
>> 10 & 1 % 10 is considered true therefore 1 & 1 = 1
ans =
1
>> 10 & 0 % 1 & 0 = 0
ans =
0
```

■

■ **Example 6.6**

```
>> ( 3 == 5 ) & ( 3 == ( 2 + 1 ) )
ans =
```

```
0
>> ( 3 == 5 ) | ( 3 == ( 2 + 1 ) )
ans =
1
```

■

6.2.2 Matrix comparison:

In MATLAB, matrix comparison involves element-wise operations where matrices are compared to each other or to scalar values. The result of these comparisons is a matrix of logical values (`true` or `false`), where each element corresponds to the result of comparing the elements of the matrices at the same position.
The comparison of vectors and matrices differs somewhat from scalars, hence the usefulness of the two functions '**isequal**' and '**isempty**' (which allow to give a concise answer for comparison).

Function	Description
Isequal	tests whether two (or more) matrices are equal (having the same elements everywhere). Returns 1 if so, and 0 otherwise.
Isempty	tests if a matrix is empty (contains no elements). Returns 1 if it is, and 0 otherwise.

■ **Example 6.7** To better perceive the impact of these functions follow the following example:

```
>> A=[5,2;-1,3] % create the matrix A
A =
 5   2
-1   3
>> B=[5,1;0,3] % create the matrix B
B =
5   1
0   3
>> A==B % test whether A=B ? (1 or 0 depending on the position)
ans =
1   0
0   1
>> isequal(A,B) % Test if A and B are equal (the same)
ans =
0
>> C=[] ; % Create the empty matrix C
>> isempty(C) % Test if C is empty (true = 1)
ans =
1
>> isempty(A) % Test if A is empty (displays false = 0)
ans =
0
```

■

6.3 Flow control structures

Flow control structures are programming constructs used to define and manage the order of execution of tasks within a program. They allow the implementation of different operations

depending on the state of program data or enable the repetition of specific processes through loops. MATLAB provides eight primary flow control structures:

- **if**
- **switch**
- **for**
- **while**
- **continue**
- **break**
- **try-catch**
- **return**

6.3.1 The if statement:

The **if** statement is the simplest and most widely used flow control structure. It guides the execution of the program according to the logical value of a condition.Its general syntax is as follows:

```
if (condition)
  instruction_1
  instruction_2
  ⋮
  instruction_N
end
```

or

```
if (condition)
  instruction set 1
else
  instruction set 2
end
```

If the condition is evaluated to true, the instructions between the `if` and the `end` will be executed; otherwise, they will not be (or if an `else` exists, the instructions between the `else` and the `end` will be executed).
If it is necessary to check multiple conditions instead of just one, you can use `else if` clauses for each new condition, and at the end, you can put an `else` in case no condition has been evaluated as true. Here is the general syntax:

```
if (expression_1)
  set of instructions 1
else if (expression_2)
  set of instructions 2
  ⋮
else if (expression_n)
  set of instructions n
else
  set of instructions if all expressions were false
end
```

■ **Example 6.8** For example, the following program defines you according to your age:

```
function[]=Age()
%Program defines you according to your age
age = input('Enter your age : '); ...
if (age <2)
 disp('You are a fool')
 elseif (age <13)
 disp('You are a child')
 elseif (age < 18)
```

```
disp ('You are an adolescent')
elseif (age <60)
disp ('You are unadulterated')
else
disp ('You are an old man')
end
```

■

As you can see, writing a Matlab program directly after the command prompt (the prompt ») is a bit unpleasant and annoying.

A more convenient method is to write the program to a separate file, and call that program (if necessary) by typing the file name in the command prompt.
This approach is defined in Matlab by M-Files, which are files that can contain data, programs (scripts) or functions that we develop.

To create an M-Files simply type the command edit, or simply go to the menu: File New MFiles.

In any case an editing window like this will appear:

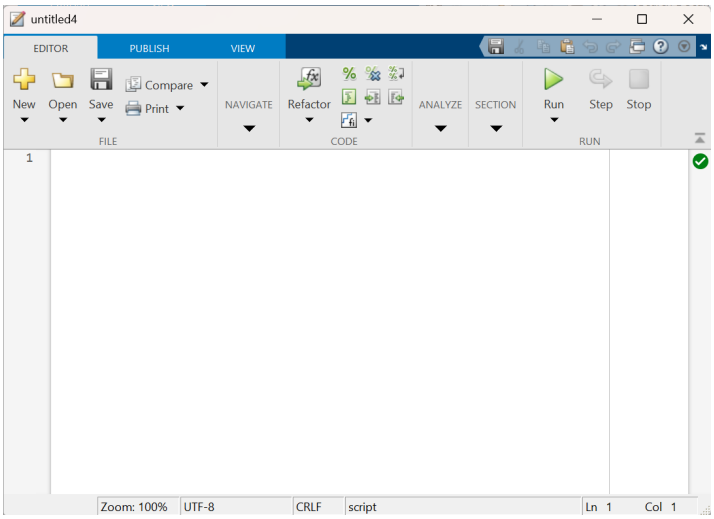


Figure 6.3.1: *Editing window*

All you have to do is write your program in this window and save it with a name (for example: '**Premier_Programme.m**'). It is reported that the extension of the M-Files files is always '**.m**'. Now, if we want to run our program, just go to the usual command prompt (») and then type the name of our file (without the '**.m**') like this:

```
>> Premier_Programme
```

To return to the editing window (after closing it) simply enter the command :

```
>> edit Premier_Programme
```

Exercise 6.1 Let's create a program that finds the roots of a second-degree equation designated

by: $ax^2+bx+c=0$. Here is the M-File that contains the program (it is saved with the name '**Equation2deg.m**'). ■

Solution 6.1

```
% Program to solve the equation a*x^2 + b*x + c = 0
a = input('Enter the value of a: '); % read a
b = input('Enter the value of b: '); % read b
c = input('Enter the value of c: '); % read c

delta = b^2 - 4*a*c; % Compute the discriminant

if delta < 0
    disp('No real solution') % No solution
elseif delta == 0
    disp('Double root:') % One double root
    x = -b / (2*a)
else
    disp('Two distinct solutions:') % Two different roots
    x1 = (-b + sqrt(delta)) / (2*a)
    x2 = (-b - sqrt(delta)) / (2*a)
end
```

If we want to run the program, just type the name of the program:

```
>> Equation2deg
Enter the value of a : -2
Enter the value of b : 1
Enter the value of c : 3
Two solutions:
x1 =
 -1
x2 =
 1.5000
```

Note 6.1 There is the predefined solve function in Matlab to find the roots of an equation (and much more). If we want to apply it to our example, just write:

```
>> solve('-2*x^2+x+3=0','x')
years =
-1
3/2
```

6.3.2 The switch statement:

The **switch statement** executes groups of statements based on the value of a variable or expression. Each group is associated with a `case` clause that defines whether or not this group should be executed according to the equality of the value of this variable with the evaluation result of the `switch` expression.
If not all cases have been satisfied, it is possible to add an `otherwise` clause that will be executed only if no case is executed.

Therefore, the general form of this instruction is:

```
switch (expression)
  case value_1
    Instruction group 1
  case value_2
    Instruction group 2
  ⋮
  case value_n
    Instruction group n
  otherwise
    Package instructions where the boxes have failed
end
```

■ **Example 6.9**

```
x = input ('Enter a number: ') ;
switch(x)
case 0
par ('x = 0 ')
case 10
par('x = 10 ')
case 100
par('x = 100 ')
otherwise
par('x n is not 0 or 10 or 100')
end
```

The execution will give:

```
Enter a number : 50
x is not 0 or 10 or 100
```

■

6.3.3 The for statement:

The **for** statement repeats the execution of a group of instructions a specified number of times. It has the following general form:

```
for  variable = expression_vector
     instruction group
end
```

`vecteur` The `expression_vector` corresponds to the definition of a vector: `start: not: end` or `start:end`

The variable will go through all the elements of the vector defined by the expression, and for each it will execute the group of instructions.

■ **Example 6.10** In the following table, we know three forms of the for statement with the Matlab result:

The instruction for	`for i = 1 : 4`
	`  j = i * 2;`
	`  disp(j)`
	`end`
		`for i = 1 : 2 : 4`
		`  j = i * 2;`
		`  disp(j)`
		`end`
			`for i = [1, 4, 7]`
			`  j = i * 2;`
			`  disp(j)`
			`end`

The result of the execution	2 4 6 8	2 6	2 8 14

■

6.3.4 The while statement:

The **while** statement repeats the execution of a group of statements an indeterminate number of times depending on the value of a logical condition. It has the following general form:

```
while  (condition)
    set of instructions
end
```

As long as the expression of **while** is evaluated to true, the instructions set will run in a loop.

■ **Example 6.11**

```
1. a=1 ;
   while (a~=0)
   a = input ('Enter unnombre (0 to finish) : ') ;
   end
```

This program asks the user to enter a number. If this number is not equal to 0 then the loop repeats, otherwise (if the given value is 0) then the program stops.

```
2. count = 1;
   while count <= 5
   disp(['Count is: ', num2str(count)]);
   count = count + 1;
   end
```

This program repeatedly displays the current value of the variable count while incrementing it by 1 in each iteration, until count reaches 5. Specifically, it prints "Count is: " followed by the current value of count. ■

R In MATLAB, you can easily replace a for loop with a while loop by handling the initialization, condition, and increment manually. Here's an example to illustrate how this can be done.

■ **Example 6.12** Example of a for loop in MATLAB:

```
a = 16; % Example value
x = a;
for i = 1:10 % Perform 10 iterations
 x = 0.5 * (x + a / x);
end
disp(x)
```

This is an implementation of Newton's method for finding the square root with 10 iterations using a for loop.
Replacing the for loop with a while loop in MATLAB:

```
a = 16; % Example value
x = a;
i = 1; % Initialization of iteration counter
while i <= 10 % Perform 10 iterations
 x = 0.5 * (x + a / x);
 i = i + 1; % Increment the counter manually
end
disp(x)
```

■

6.4 Summary the control structures:

MATLAB provides several **flow control structures** that manage the execution of code based on conditions or repetitive operations. These include conditional statements (`if`, `else`, `elseif`), looping constructs (`for`, `while`), and error handling using `try` and `catch`. Such structures are fundamental for developing flexible, robust, and efficient programs.
We gives now a glimpse of:

1. **break** and **continue** Statements
   - **break:** Exits the loop completely, even if the loop condition is still true.

     ■ **Example 6.13 Using break:**

     ```
     for i = 1:10
        if i == 5
          break; % Exit the loop when i equals 5
        end
        disp(i);
     end
     ```

     ■

   - **continue:** Skips the current iteration and moves to the next iteration of the loop.

     ■ **Example 6.14 Using continue:**

     ```
     for i = 1:10
        if mod(i, 2) == 0
          continue; % Skip the rest of the loop for even numbers
        end
        disp(i); % Display only odd numbers end
     ```

     ■

2. **try** and **catch** Statements The `try` and `catch` blocks are used for **error handling**. If an error occurs within the `try` block, control is automatically transferred to the `catch` block, allowing the program to handle the error gracefully without terminating execution.
   Syntax:

   ```
   try
   ```

```
    % Code that may cause an error
catch exception
    % Code to handle the error
end
```

■ **Example 6.15**

```
1- try
   A = [1, 2, 3];
     disp(A(4)); % This will cause an error because there is no 4th
     element
   catch exception
     disp('An error occurred:');
     disp('Index exceeds the number of array elements. Index must not exceed 3.');
   end

2- try
     % Attempt to divide by zero
     result = 10 / 0;
     disp(['Result: ', num2str(result)]);
     element
  catch exception
     %Handle the error gracefully
     disp('Error occurred: Division by zero is  not allowed.');
  end
```

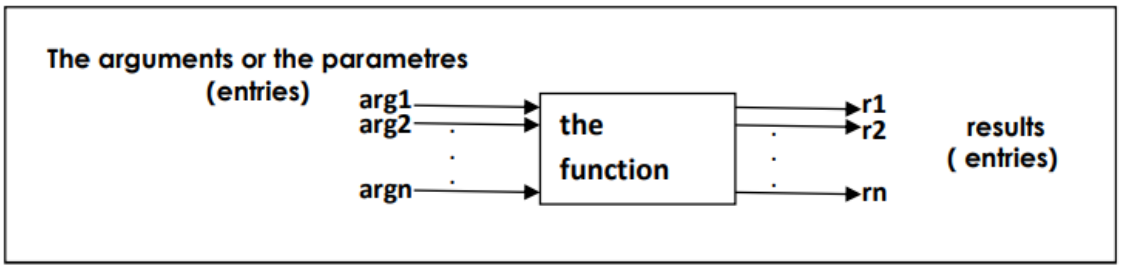
■

6.5 The functions:

The Functions:

There is a conceptual difference between functions in computer science and in mathematics:

1. In computer science, a function is a routine (or sub-program) that accepts one or more arguments (parameters) and returns a result. It is used to perform specific tasks or calculations, allowing code to be reused efficiently.



2. In mathematics: A function f is a relationship that assigns to each value x at most one value $f(x)$.

6.5.1 Creating a function in an M-Files:

Matlab contains a large number of predefined functions such as sin, cos, sqrt, sum, ... etc. and it is possible to create our own functions by writing their source codes in M-Files (with the same function name) respecting the following syntax:

```
function [r1, r2, …, rn] = nom_fonction (arg1, arg2, …, argn)

      % The body of the function
      . . .
      r1 = . . .   % the value returned for r1
      r2 = . . .   % The value returned for r2
      . . .
      rn = . . .   % the value returned for rn
end                % The end is optional
```

Or: $r_1, ..., r_n$ are the values returned, and $arg_1, ..., arg_n$ are the arguments.

Exercise 6.2 Write a function that calculates the square root of a number by the Newton method. ■

Solution 6.2

```
function r =racine(numbre)
   r = numbre/2;
   precision = 6;
   for i = 1:precision
      r = (r + numbre ./ r) / 2;
   end
end
```

Execution:

```
>> x = racine(4)
 x =
  2
>> x = racine(196)
 x =
  14.0000
>> x = racine([16,144,9,5])
x =
  4.0000   12.0000   3.0000   2.2361
>> x = racine([16, 14, 81; 36 25 12; 30, 400, 50])
x =
    4.0000    3.7417    9.0000
    6.0000    5.0000    3.4641
    5.4772   20.0001    7.0711
```

6.5.2 Comparison between a program is a function:

Program	Function

`a = input('Enter a positive number:...');`
`x = a/2;`
`Precision = 6;`
`for i = 1:precision`
`  x = (x + a ./ x) / 2;`
`end`
`disp(x)`
	`function r =root(number)`
	`r = number /2;`
	`Precision = 6;`
	`for i = 1: Precision`
	`  r = (r + number ./ r) / 2;`
	`end`

Execution:
`>> root`
`Enter a positive number: 16`
`  4`
	Execution:
	`>> root(16)`
	`  ans =`
	`  4`

One cannot write expressions such as: `>> 2* root + 4`	One cannot write expressions such as: `>> 2* root(x) + 4`

Exercise 6.3 Write a script in Matlab, that asks the user for a number, and then informs them whether the number is *positive* or *negative* (ignore the case where the number is zero). ■

Exercise 6.4 Write a MATLAB script that asks the user to enter two numbers, and then displays whether their *product* is positive or negative — **without actually computing the product**. (The case where the product is zero may be ignored.) ■

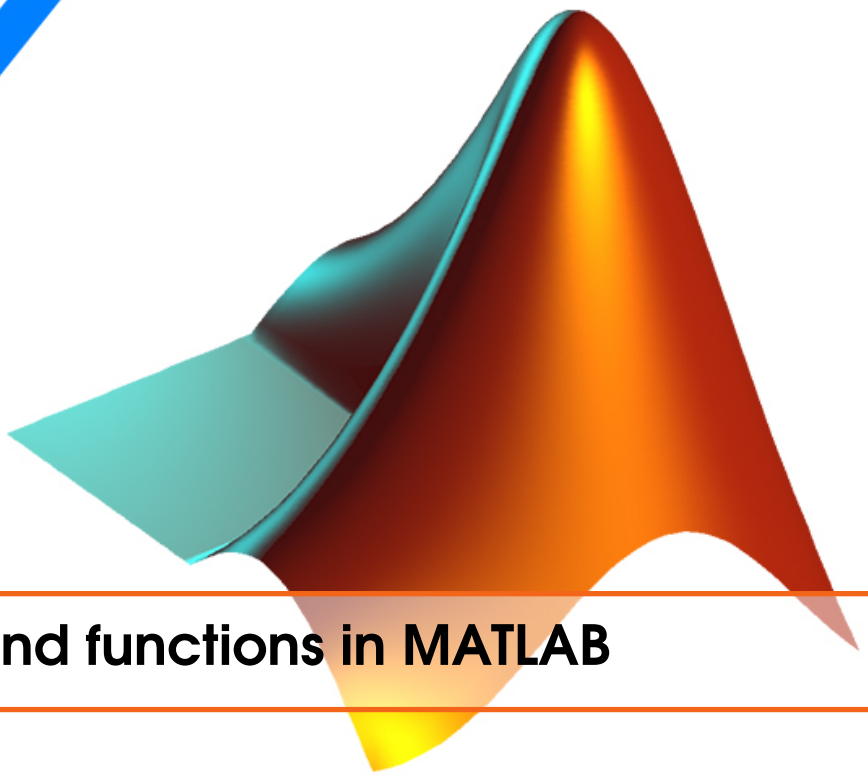
Exercise 6.5 Write two MATLAB scripts that calculate the sum of the first n natural numbers $(1,2,3,...,n)$:

1. Using a for loop.
2. Using a while loop.

■

CYDOMEDIA

WHAT IS
MATLAB?
MATLAB PLOT AND
MATLAB FUNCTIONS



7. Polynomials and functions in MATLAB

7.1 Polynomials in MATLAB:

MATLAB provides powerful tools for working with polynomials, including the ability to define polynomials, find their roots (zeros), and perform various operations like addition, multiplication, and division.

7.1.1 Polynomials:

In MATLAB, polynomials can be represented using vectors, where each element corresponds to a coefficient of the polynomial. The vector elements are ordered in descending powers of the variable.

For example, the polynomial $p(x) = 2x^3 + 3x^2 - x + 5$ can be represented as:

```
>> p = [4  6  -2  10]; % Polynomial 4x^3 + 6x^2 - 2x + 10
```

Here, the vector $[4\ 6\ -2\ 10]$ represents the polynomial $4x^3 + 6x^2 - 2x + 10$ or $4x \wedge 3 + 6x \wedge 2 - 2x + 10$, with the first element being the coefficient of x^3, the second element being the coefficient of x^2, and so on.

7.1.2 Polynomial Zeros:

You can find the roots (zeros) of a polynomial using the `roots()` function, which calculates the values of x for which the polynomial equals zero.

For example, to find the zeros of the polynomial $p(x) = 2x^3 + 3x^2 - x + 5$:

```
>> zeros_p = roots(p); % Finds the roots of the polynomial p.
```

This will return the values of x that satisfy $p(x) = 0$.

7.1.3 Polynomial Operations:

MATLAB allows you to perform several operations on polynomials, such as addition, subtraction, multiplication, division, derivative, and integral.

- **Polynomial Addition and Subtraction:** Polynomials can be added or subtracted directly by adding or subtracting their coefficient vectors.

```
>> p1 = [2 3 -1 5]; % First polynomial.
>> p2 = [1 -4 2]; % Second polynomial.
>> sum_p = p1 + p2; % Polynomial addition.
>> diff_p = p1 - p2; % Polynomial subtraction.
```

- **Polynomial Multiplication:** Polynomials can be multiplied using the `conv()` function, which performs the convolution of two polynomials.

```
>> p1 = [1 2];      % x + 2
>> p2 = [1 -3];     % x - 3
>> product = conv(p1, p2)    % Multiplies p1 and p2.
product =
     1    -1    -6
```

- **Polynomial Division:** To divide one polynomial by another, use the `deconv()` function, which returns both the quotient and the remainder.

```
>> p = [1 -6 11 -6];
q = [1 -2];
>> [quotient, remainder] = deconv(p, q)   % Divides p by q.
quotient =
 1    -4     3
remainder =
     0     0     0     0
```

- **Derivative and Integral of a Polynomial:** Polynomials can be differentiated and integrated using the MATLAB functions `polyder()` and `polyint()`, respectively.

```
>> p = [3 0 -5 2];
dp = polyder(p)
dp =
     9     0    -5
>> intp = polyint(p)   % constant of integration = 0
intp =
    0.7500         0   -2.5000    2.0000         0
>> intp2 = polyint(p, 7)   % constant = 7
intp2 =
    0.7500         0   -2.5000    2.0000    7.0000
```

- **Creating a Polynomial from Roots:** This can be done using the `poly` function in MATLAB.

```
>> r = [1 2 3];
>> p = poly(r)   % gives coefficients of (x-1)(x-2)(x-3)
p =
     1    -6    11    -6
```

- **Evaluating Polynomials:** A polynomial can be evaluated at specific points using the `polyval` function in MATLAB.

```
>> p = [3 0 -5 2];
>> value = polyval(p, 4);   % evaluates p(4)
value =
     6
```

Summary

In MATLAB, you can:

- Represent polynomials using coefficient vectors.
- Find the zeros (roots) of a polynomial using `roots()`.
- Perform various polynomial operations like addition, subtraction, multiplication (`conv()`), and division (`deconv()`).

These tools make MATLAB a powerful environment for working with polynomials in numerical and symbolic computations.

7.2 Function

- **Evaluation:** The 'feval' function allows you to evaluate a function.

  ■ **Example 7.1**

```
>> f = @(x) 2.*x.^2+1;
>> feval(f,1)

ans =
3
>> f(1)
ans =

 3
```

■

- **Curve:** In addition to the 'plot' function, the 'fplot' function allows you to plot the curve of a function between two values.

  ■ **Example 7.2**

```
>> f = @(x) 2. * x.^2 + 1;
>> fplot(f, [0 5])
>> fplot('fct', [0 5]) %If f is defined in an .m file use the following syntax:
>>fplot(@fct, [0 5])
```

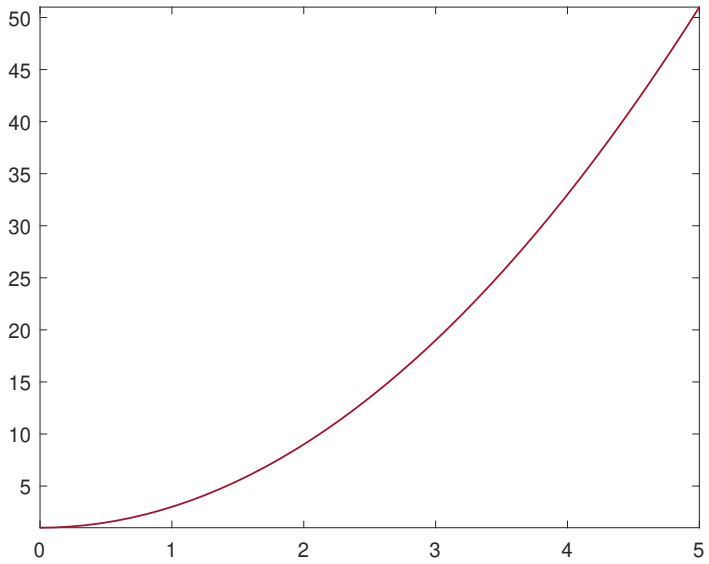


Figure 7.2.1: *The curve of the function f between 0 and 5.*

■

- **Zero of a Function:** The `fzero` function is used to compute the zeros of a function of a single variable. Specifically, `fzero(f, a)` searches for a root of the function `f` near the initial guess `a`.

  ■ **Example 7.3** $g(x) = x + 2e^{-x} - 3$

```
 function res = g(x)
 res = x + exp(-x)-3;
return
```

Two zeros: one near -01.5 and the other near 3.

```
>>fplot (@g,[-1 5])
>>fzero(@g,-0.5)
 ans =
 - 0.5831

>>fzero(@g,3)
 ans =

 2.8887
```

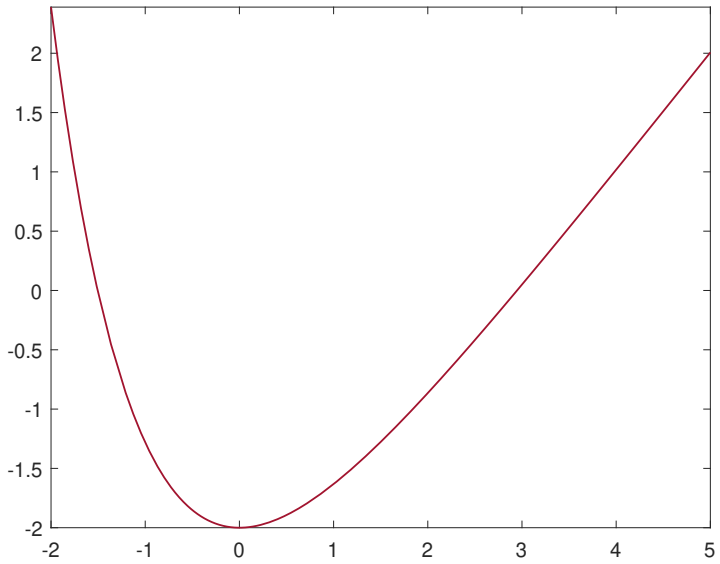


Figure 7.2.2: *The curve of the function g between* -3 *and* 5.

■

- **Minimum of a function:**

  ■ **Example 7.4** `>> fminbnd(@cos,0,4)`
  `ans =`

  `3.1416`
  $y = 1 - xe^{-x}$

  ```
  function res = h(x)
  res=1-x.*exp(-x);
  return
  ```

  To find the value of x that gives the minimum value of y in the interval [0,5]:

  ```
  >> x = fminbnd(@h,0,5)
  x=

   1.0000
  ```

  To find the minimum value of y:

  ```
  >>y = h(x)
  y

   = 0.6321
  ```

  ■