CHAPTER 4
Programming in MATLAB

CHAPITRE 4 : Programming in MATLAB

4.1. Introduction

This chapter covers the essential concepts for programming in MATLAB, focusing on the use of scripts, functions, and control statements. Programming enables task automation and the development of complex algorithms, making it a crucial skill in engineering and scientific computing. By studying this chapter, you will gain the ability to write and execute MATLAB code efficiently, allowing you to solve more complex problems.

After completing this chapter, you should be able to:

- Understand and use MATLAB's arithmetic, logical, and special operators.
- Develop and manage scripts and functions using M-files.
- Utilize control structures such as loops and conditional statements to manage program execution.
- Write structured and reusable code to solve computational problems efficiently in MATLAB.

4.2. Arithmetic, Logical, and Special Operators

This section presents the most commonly used operators, along with their syntax and applications.

4.2.1. Arithmetic Operators

Arithmetic operators perform mathematical computations on numbers or matrices.

4.2.1. Arithmetic Operators

Arithmetic operators perform mathematical computations on numbers or matrices.

- **Addition (+):** Adds two numbers or matrices.

  Matlab

```
a = 5 + 3;   % Result: 8
A = [1, 2; 3, 4] + [5, 6; 7, 8];  % Result: [6, 8; 10, 12]
```

- **Subtraction (-):** Subtracts one number or matrix from another.

  Matlab

```
b = 10 - 4;  % Result: 6
B = [5, 6; 7, 8] - [1, 2; 3, 4];  % Result: [4, 4; 4, 4]
```

- **Multiplication (*):** Multiplies two matrices (matrix multiplication) or a scalar by a matrix.

  Matlab

```
c = 4 * 2;   % Result: 8
C = [1, 2; 3, 4] * [2; 1];  % Result: [4; 10]
```

- **Element-wise multiplication (.*):** Multiplies matrices element-wise.

Matlab

```
D = [1, 2; 3, 4] .* [2, 2; 2, 2];  % Result: [2, 4; 6, 8]
```

- **Division (/):** Right matrix division.

Matlab

```
e = 8 / 2;   % Result: 4
E = [2, 4; 6, 8] / 2;  % Result: [1, 2; 3, 4]
```

- **Element-wise division (./):** Divides matrices element-wise.

Matlab

```
F = [4, 9; 16, 25] ./ [2, 3; 4, 5];  % Result: [2, 3; 4, 5]
```

- **Exponentiation (^):** Raises a number or a matrix to a power (matrix power for matrices).

Matlab

```
g = 2^3;   % Result: 8
G = [1, 2; 3, 4]^2;  % Result: [7, 10; 15, 22]
```

- **Element-wise exponentiation (.^):** Raises each element of a matrix to a power.

Matlab

```
H = [1, 2; 3, 4].^2;  % Result: [1, 4; 9, 16]
```

- **Modulo (mod):** Returns the remainder of a division.

Matlab

```
r = mod(10, 3);  % Result: 1
```

- **Integer division (floor, ceil, fix, round):** Functions for rounding division results.

Matlab

```
f = floor(5/2);  % Result: 2 (rounds down)
c = ceil(5/2);   % Result: 3 (rounds up)
```

4.2.2. Logical Operators

Logical operators evaluate Boolean expressions. The results are either true (1) or false (0).

- **Logical AND (&& for scalars, & for matrices):** Returns true if both conditions are true.

Matlab

```
a = true && false;  % Result: 0 (false)
A = [true, false; true, true] & [true, true; false, true];  % Result: [1, 0; 0, 1]
```

- **Logical OR (|| for scalars, | for matrices):** Returns true if at least one condition is true.

```
b = true || false;  % Result: 1 (true)
```

5. *B = [true, false; false, false] | [false, true; true, false]; % Result: [1, 1; 1, 0]*

- **Logical NOT (~):** Inverts a Boolean value.

6. *c = ~true; % Result: 0 (false)*
7. *C = ~[true, false]; % Result: [0, 1]*

- **Equality (==):** Checks if two values are equal.

8. *d = (5 == 5); % Result: 1 (true)*

- **Inequality (~=):** Checks if two values are different.

9. *e = (5 ~= 3); % Result: 1 (true)*

4.2.3. Special Characters

Special characters perform specific tasks such as matrix indexing, string concatenation, and special matrix creation.

- **Colon (:):** Creates vectors, indexes matrices, or specifies ranges.

10. *matlab*
11. *x = 1:5; % Result: [1, 2, 3, 4, 5]*
12. *A = [1, 2; 3, 4; 5, 6];*
13. *row1 = A(1, :); % Retrieves the first row: [1, 2]*

- **Semicolon (;):** Separates rows in a matrix or suppresses command output.

14. *B = [1, 2; 3, 4]; % Creates a 2x2 matrix*
15. *C = 5 + 5; % Result not displayed*

- **Brackets ([]):** Used to create matrices and arrays.

16. *D = [1, 2, 3]; % Row vector*
17. *E = [1; 2; 3]; % Column vector*

- **Parentheses (()):** Used for indexing matrices, function calls, and expression grouping.

18. *F = [10, 20, 30];*
19. *G = F(2); % Retrieves the second element: 20*

- **Apostrophe ('):** Used for matrix transposition.

20. *H = [1, 2, 3];*
21. *I = H'; % Result: [1; 2; 3] (column vector)*

- **Dot (.):** Used for element-wise operations on matrices.

22. *J = [1, 2; 3, 4];*
23. *K = J.^2; % Squares each element*

4.3. M-Files (MATLAB Scripts and Functions)

M-Files (or .m files) are scripts and functions created in MATLAB. They contain MATLAB code that can be executed directly within the MATLAB environment. M-Files form the foundation of MATLAB programming, allowing for code structuring, automation, and reuse.

4.3.1. Types of M-Files

There are two main types of M-Files:

- **Scripts:** An M-File containing a sequence of MATLAB commands. Scripts do not accept inputs or return explicit outputs, but they can operate on variables present in the MATLAB workspace.
- **Functions:** An M-File that starts with a function declaration. Functions accept inputs and can return outputs. They are independent of the workspace and have their own variable scope.

4.3.2. Creating M-Files

4.3.2.1. Creating a Script

1. Open MATLAB.
2. In the "Home" tab, click "New Script."
3. Write your sequence of MATLAB commands in the text editor.
4. Save the file with the extension .m.

Example of a script:

Matlab

```
1. % Script to calculate the average of a list of numbers
2. numbers = [10, 20, 30, 40, 50];
3. average = mean(numbers);
4. disp(['The average is: ', num2str(average)]);
```

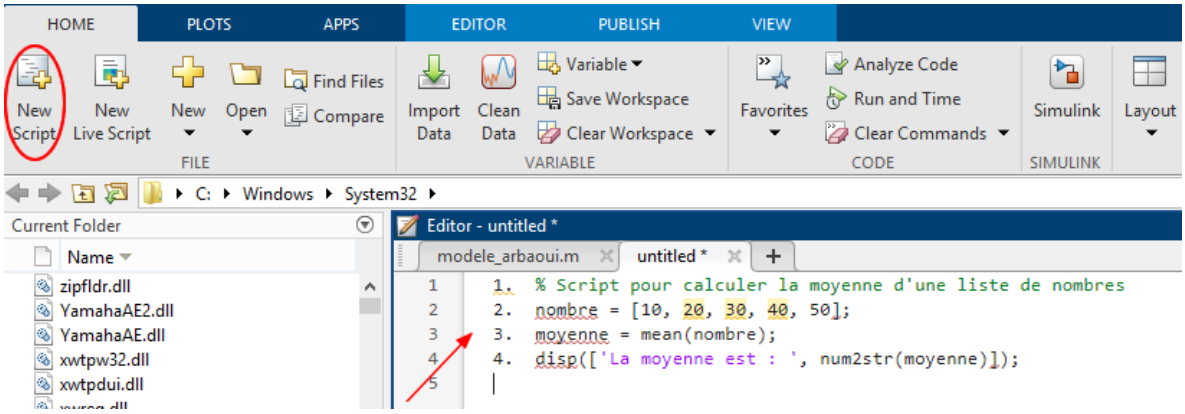

Figure 4.1: Creating a Script

4.3.2.2. Creating a Function

1. Open MATLAB.
2. In the "Home" tab, click "New Script."
3. Write the function definition, starting with the function keyword.
4. Save the file with a name matching the function name, using the .m extension.

Example of a function:

Matlab

```
1. function y = square(x) % Function that calculates the square of a number
2. y = x^2;
3. end
```

4.3.3. Executing M-Files

4.3.3.1. Running a Script

To execute a script, type the filename (without the .m extension) in the MATLAB command window and press **Enter**.

Matlab

```
1. my_script  % Runs the script named my_script.m
```

4.3.3.2. Running a Function

To execute a function, type the function name followed by the required input arguments in the MATLAB command window.

Matlab

```
1. result = square(5);  % Calls the function square with input 5
```

4.3.4. Advantages of Using M-Files

- **Code Organization:** M-Files allow the separation of code into logical modules, making it easier to read, debug, and maintain.
- **Code Reusability:** Functions in M-Files can be reused in other scripts or programs.
- **Automation:** Scripts help automate repetitive tasks, reducing human errors.

4.3.5. Comments and Documentation

Documenting your code is important to explain its purpose. In MATLAB, comments are added using the % symbol. Everything following % on a line is ignored by MATLAB during execution.

Example of a well-commented script:

Matlab

```
1. % Script to calculate the sum of squares from 1 to 10
2. sum_squares = 0;  % Initialize sum
3. for i = 1:10
4. sum_squares = sum_squares + i^2;  % Add the square of i to the sum
5. end
6. disp(['The sum of squares is: ', num2str(sum_squares)]);
```

4.3.6. Anonymous Functions and Subfunctions

- **Anonymous Functions:** Simple functions that are defined directly in the code without needing separate M-Files.

  Matlab

```
1. square = @(x) x^2;
2. result = square(5);  % Output: 25
```

- **Subfunctions:** Additional functions defined in the same M-File after the main function. They can only be accessed by the main function or other subfunctions within the same file.

Example:

Matlab

```
1. function main_function()
2. disp('Calling the main function');
3. secondary_function();  % Calls the subfunction
4. end

5. function secondary_function()
6. disp('Calling the secondary function');
7. end
```

4.3.7. Best Practices for M-Files

- **Use Meaningful File Names:** The file name should reflect its function or content.
- **Modularity:** Divide complex tasks into smaller, simpler functions.
- **Documentation:** Use comments to explain the code, especially for complex sections.
- **Test Your Functions:** Ensure your functions produce expected results before integrating them into larger projects.

4.4. Control Statements

Control statements in MATLAB allow controlling the execution flow of a program based on conditions, loops, or other logical structures. They are essential for writing programs that respond to different situations or repeat tasks efficiently.

4.4.1. FOR and WHILE Loops

Loops are used to repeat a block of code multiple times.

4.4.1.1. FOR Loop

The for loop in MATLAB is a powerful tool for executing repetitive operations with precise control over the number of iterations. It is widely used, from iterating over vectors to performing complex matrix manipulations. Mastering the for loop allows writing more efficient and readable code.

4.4.1.1.1. Syntax of the FOR Loop

Matlab

```
1. CopierModifier
2. for variable = start:increment:end
3. % Instructions to repeat
4. end
```

Where:

- **variable:** The iteration variable that successively takes each value in the specified range start:increment:end.
- **start:** The initial value of the variable.

- **increment:** The step by which the variable increases in each iteration (default increment is 1).
- **end:** The final value; the loop stops once the variable reaches or exceeds this value.
- **Instructions to repeat:** The code that will be executed at each iteration.

4.4.1.1.2. Examples of FOR Loops

Simple Loop:

This example prints numbers from 1 to 5.

Matlab

```
for i = 1:5
disp(['Iteration: ', num2str(i)]);
end
```

Output:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

Loop with Step Size of 2:

This example prints numbers from 1 to 9 using a step size of 2.

Matlab

```
for i = 1:2:9
disp(['Value of i: ', num2str(i)]);
end
```

Output:

```
Value of i: 1
Value of i: 3
Value of i: 5
Value of i: 7
Value of i: 9
```

Reverse Loop:

A loop that counts down from 5 to 1.

Matlab

```
for i = 5:-1:1
disp(['Counter: ', num2str(i)]);
end
```

Output:

4. *Counter: 5*
5. *Counter: 4*
6. *Counter: 3*
7. *Counter: 2*
8. *Counter: 1*

Loop with a Vector of Values:

A loop that iterates over specific values in a vector.

Matlab

```
values = [10, 20, 30, 40];
for i = values
disp(['Value: ', num2str(i)]);
end
```

Output:

5. *Value: 10*
6. *Value: 20*
7. *Value: 30*
8. *Value: 40*

Loop to Calculate a Sum:

This example computes the sum of the squares of numbers from 1 to 5.

Matlab

```
sum_squares = 0;
for i = 1:5
sum_squares = sum_squares + i^2;
end
disp(['The sum of squares is: ', num2str(sum_squares)]);
```

Output:

6. *The sum of squares is: 55*

Using Nested FOR Loops:

Nested loops are useful for iterating through matrices or performing multidimensional operations.

Example: Iterating Through a Matrix

Matlab

```
matrix = [1, 2, 3; 4, 5, 6; 7, 8, 9];
for i = 1:3
for j = 1:3
fprintf('Element (%d,%d) = %d\n', i, j, matrix(i, j));
end
end
```

Output:

```
Element (1,1) = 1
Element (1,2) = 2
Element (1,3) = 3
Element (2,1) = 4
Element (2,2) = 5
Element (2,3) = 6
Element (3,1) = 7
Element (3,2) = 8
Element (3,3) = 9
```

4.4.1.2. WHILE Loop

A while loop in MATLAB is used to repeat a block of code as long as a specified condition remains true. Unlike for loops, where the number of iterations is known beforehand, while loops continue execution until the condition becomes false.

4.4.1.2.1. Syntax of the WHILE Loop

Matlab

```
while condition
% Instructions to repeat
end
```

- **condition:** A logical expression evaluated before each iteration. If the condition is true (nonzero), MATLAB executes the loop's instructions. If false, the loop stops.
- **Instructions to repeat:** The block of code executed as long as the condition remains true.

4.4.1.2.2. Examples of WHILE Loops

Simple WHILE Loop:

This example runs the loop as long as i is less than or equal to 5, incrementing i by 1 each time.

Matlab

```
i = 1;
while i <= 5
disp(['Value of i: ', num2str(i)]);
i = i + 1;
end
```

Output:

```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

WHILE Loop with a Complex Condition:

This example adds the square of i to sum_squares in each iteration until sum_squares reaches or exceeds 100.

Matlab

```
sum_squares = 0;
i = 1;
while sum_squares < 100
sum_squares = sum_squares + i^2;
disp(['i = ', num2str(i), ', sum = ', num2str(sum_squares)]);
i = i + 1;
end
```

Output:

```
i = 1, sum = 1
i = 2, sum = 5
i = 3, sum = 14
i = 4, sum = 30
i = 5, sum = 55
i = 6, sum = 91
i = 7, sum = 140
```

Infinite Loop (and How to Avoid It):

A while loop without a proper termination condition can cause an infinite loop, making the code run indefinitely.

Matlab

```
i = 1;
while i > 0
disp(['i = ', num2str(i)]);
% This loop never stops because `i` is always positive
end
```

To prevent this, ensure that the condition eventually becomes false, or use a break statement to exit the loop:

Matlab

```
i = 1;
while true
disp(['i = ', num2str(i)]);
i = i + 1;
if i > 10
break; % Exits the loop when i exceeds 10
end
end
```

Using CONTINUE in a WHILE Loop:

The continue statement skips the remaining instructions in the loop and jumps to the next iteration.

Example: The loop skips iterations where i is even.

Matlab

```
1. i = 0;
2. while i < 10
3. i = i + 1;
4. if mod(i, 2) == 0
5. continue;  % Skips display for even numbers
6. end
7. disp(['i = ', num2str(i)]);
8. end
```

Output:

```
9. i = 1
10. i = 3
11. i = 5
12. i = 7
13. i = 9
```

4.4.2. Error Handling Instructions

Error handling instructions in MATLAB help manage errors that occur during program execution. These instructions prevent abrupt program termination and allow for appropriate responses when an error arises.

4.4.2.1. Using try and catch

The try-catch structure is used to capture errors that occur within a block of code. If an error occurs inside the try block, MATLAB immediately switches to the catch block to execute error-handling code.

4.4.2.1.1. Syntax

Matlab

```
1. try
2. % Code that may potentially cause an error
3. catch
4. % Code executed in case of an error
5. end
```

Example:

In this example, the program attempts to divide a number by zero, which generates an error. Instead of stopping execution, a custom error message is displayed.

Matlab

```
1. try
2. result = 10 / 0;  % This generates a division-by-zero error
3. catch
4. disp('An error occurred: division by zero.');
5. end
```

Output:

```
6. An error occurred: division by zero.
```

Example: Capturing Detailed Error Messages

You can capture detailed error information using a second argument in catch.

Matlab

```
1. try
2. x = [1, 2, 3];
3. y = x(4);  % This generates an error since index 4 is out of bounds
4. catch ME
5. disp('An error occurred:');
6. disp(ME.message);  % Displays MATLAB's error message
7. end
```

Output:

```
8. An error occurred:
9. Index exceeds the number of array elements (3).
```

In this example, ME is a structure containing details about the error, including the error message, identifier, and execution stack.

4.4.2.2. Using error

The error command explicitly generates an error in MATLAB. This can be useful for handling custom error conditions.

4.4.2.2.1. Syntax

Matlab

```
1. error('Custom error message');
```

Example:

In this example, an error is generated if the input variable is not positive.

Matlab

```
2. function checkPositiveNumber(x)
3. if x <= 0
4. error('The number must be positive.');
5. else
```

```
6.  disp('The number is positive.');
7.  end
8.  end

9.  % Calling the function with a negative number
10. checkPositiveNumber(-5);
```

Output:

```
11. Error using checkPositiveNumber (line 3)
12. The number must be positive.
```

4.4.2.3. Using warning

The warning function is similar to error, but it does not stop the program execution. Instead, it displays a warning message while allowing the program to continue.

4.4.2.3.1. Syntax

```
Matlab
1. warning('Custom warning message');
```

Example:

In this example, a warning is displayed if the number is negative, but the program continues execution.

```
Matlab
2.  CopierModifier
3.  function checkNumber(x)
4.  if x < 0
5.  warning('The number is negative.');
6.  end
7.  disp(['The number is: ', num2str(x)]);
8.  end

9.  % Calling the function with a negative number
10. checkNumber(-5);
```

Output:

```
11. Warning: The number is negative.
12. The number is: -5
```

4.4.2.4. Using assert

The assert function checks if a condition is true. If the condition is false, an error is generated. This is useful for input validation and debugging.

4.4.2.4.1. Syntax

Matlab

```
1. assert(condition, 'Custom error message');
```

Example:

In this example, assert ensures that b is not zero before performing division. If b is zero, an error is triggered.

Matlab

```
1. function divide(a, b)
2. assert(b ~= 0, 'The denominator must not be zero.');
3. result = a / b;
4. disp(['The result is: ', num2str(result)]);
5. end
6. % Calling the function with b = 0
7. divide(10, 0);
```

Output:

```
8. Error using assert (line 89)
9. The denominator must not be zero.
```

4.4.3. tic and toc in MATLAB

The tic and toc functions in MATLAB measure the execution time of a block of code. These commands are useful for evaluating algorithm performance and identifying time-consuming parts of a program.

4.4.3.1. Syntax

Matlab

```
1. tic
2. % Code to measure execution time
3. toc
```

Example: Measuring Execution Time

This example measures the time required to compute the sum of numbers from 1 to 1,000,000.

```
1. matlab
2. tic % Start timer
3. sum_total = 0;
4. for i = 1:1000000
5. sum_total = sum_total + i;
6. end
7. toc  % Stop timer and display elapsed time
```

Output:

```
8. Elapsed time is 0.013284 seconds.
```

4.4.3.2. Measuring Multiple Sections of Code

Multiple tic-toc pairs can measure different sections of a program.

```
Matlab
tic
% Section 1 of the code
pause(1);  % Simulates a calculation taking 1 second
toc

tic
% Section 2 of the code
pause(2);  % Simulates a calculation taking 2 seconds
toc
```

Output:

```
Elapsed time is 1.000678 seconds.
Elapsed time is 2.001213 seconds.
```

4.4.3.3. Storing Elapsed Time in a Variable

You can store the elapsed time in a variable by assigning toc to a variable.

```
matlab
tic
pause(1.5);  % Simulates a calculation taking 1.5 seconds
elapsed_time = toc;
disp(['Elapsed time: ', num2str(elapsed_time), ' seconds.']);
```

Output:

```
Elapsed time: 1.500345 seconds.
```

4.4.3.4. Advanced Usage: Managing Multiple Timers

MATLAB allows multiple timers to be used simultaneously by assigning identifiers to tic.

```
Matlab
t1 = tic;  % Start first timer
pause(1);  % Simulate a 1-second calculation
time1 = toc(t1);  % Stop first timer
t2 = tic;  % Start second timer
pause(2);  % Simulate a 2-second calculation
time2 = toc(t2);  % Stop second timer
disp(['Time for first block: ', num2str(time1), ' seconds.']);
disp(['Time for second block: ', num2str(time2), ' seconds.']);
```

Output:

```
Time for first block: 1.000234 seconds.
Time for second block: 2.000457 seconds.
```

4.4.4. Conditional Statements

Conditional statements in MATLAB are used to control the flow of code execution based on specific conditions. They allow different sections of code to be executed depending on variable values or states. The main conditional structures in MATLAB are if, elseif, else, and switch.

4.4.4.1. The if, elseif, else Structure

The if structure executes a block of code if a condition is true. The elseif and else blocks handle alternative conditions.

4.4.4.1.1. Syntax

Matlab

```
if condition
% Code executed if the condition is true
elseif other_condition
% Code executed if the first condition is false and this condition is true
else
% Code executed if all previous conditions are false
end
```

Simple Example:

In this example, since x is less than 10, the else block is executed.

Matlab

```
x = 5;
if x > 10
disp('x is greater than 10');
elseif x == 10
disp('x is equal to 10');
else
disp('x is less than 10');
end
```

Output:

```
x is less than 10
```

Example with Multiple Conditions:

In this example, the condition y >= 10 & y <= 20 is true, so the corresponding elseif block is executed.

Matlab

```
y = 15;
if y < 10
disp('y is less than 10');
```

```
4. elseif y >= 10 && y <= 20
5. disp('y is between 10 and 20');
6. else
7. disp('y is greater than 20');
8. end
```

Output:

```
9. y is between 10 and 20
```

4.4.4.2. The switch Structure

The switch structure is used to choose between multiple alternatives based on the value of an expression. It is often more readable than multiple if-elseif statements when handling multiple cases.

4.4.4.2.1. Syntax

Matlab

```
1. switch expression
2. case value1
3. % Code executed if expression == value1
4. case value2
5. % Code executed if expression == value2
6. otherwise
7. % Code executed if none of the values match
8. end
```

Simple Example:

In this example, the variable day has the value 'Monday', so the first case block is executed.

Matlab

```
1. day = 'Monday';
2. switch day
3. case 'Monday'
4. disp('Start of the week');
5. case 'Wednesday'
6. disp('Midweek');
7. case 'Friday'
8. disp('End of the week');
9. otherwise
10. disp('Unspecified day');
11. end
```

Output:

```
12. Start of the week
```

Example with a Numeric Value:

In this example, the value of grade is between 80 and 89, so the corresponding block is executed.

Matlab

```
1. grade = 85;
2. switch grade
3. case {90, 100}
4. disp('Excellent');
5. case {80, 89}
6. disp('Very good');
7. case {70, 79}
8. disp('Good');
9. otherwise
10. disp('Needs improvement');
11. end
```

Output:

```
12. Very good
```

4.5. Conclusion

This chapter introduced essential MATLAB programming concepts, including scripts, functions, and control statements. You have acquired skills to structure code, automate tasks, and develop algorithms to solve complex problems. These programming skills are crucial for advancing your knowledge of MATLAB, enabling you to create sophisticated simulations and analyses that will be essential in your future work in engineering and technology.