CHAPTER 2
Data Types and Variables

CHAPTER 2: Data Types and Variables

2.1. Introduction

This chapter introduces the fundamental concepts necessary to understand and manipulate data types and variables in MATLAB. Data types such as complex numbers, Boolean variables, vectors, matrices, and polynomials are essential for mathematical modeling and problem-solving. Mastering these concepts will enable you to handle a wide range of computational tasks in MATLAB, laying the groundwork for more advanced programming and analysis.

After studying this chapter, you should be able to:

- Recognize and manipulate different data types in MATLAB, such as complex numbers and Boolean variables.
- Perform manipulations and operations on vectors, matrices, and polynomials.
- Understand the importance of data types in mathematical modelling and their relevance to solving real-world engineering problems.
- Apply these concepts to efficiently structure and analyse data within MATLAB.

2.2. Data Types

In programming, data types are classifications used to indicate what kind of data can be stored and manipulated by a program. Each data type imposes restrictions on the values that a variable can take and the operations that can be performed on that variable. These data types form the basis of programming, and each language may have its own variations or names for these types, as well as additional types specific to that language.

2.2.1. Primitive Data Types

1. **Integers (int):** Represent whole numbers, such as 1, 2, … or -5, -15 …
2. **Floating-point numbers (float, double):** Represent numbers with decimal points, like 3.14, …
3. **Characters (char):** Represent a single character, such as 'a', 'B', or '9'.
4. **Booleans (bool):** Represent truth values, either true or false.
    - **Int:** Used for whole numbers, quick to manipulate, and without decimals.
    - **Float/Double:** Used for numbers with fractional parts, with double offering higher precision.
    - **Char:** For individual characters, often used in text management.
    - **Bool:** For logic, conditions, and flow control.

2.2.2. Composite Data Types

1. **Strings (string):** Represent a sequence of characters, like "Hello" or "123ABC".
2. **Arrays:** A collection of values of the same type, indexed by an integer. For example, [1, 2, 3] is a horizontal array of integers.

3. **Sets:** An unordered collection of unique values.
4. **Dictionaries (maps):** A collection of key-value pairs, where each key is unique and associated with a value.

2.2.3. Abstract Data Types

- **Objects:** In object-oriented programming, an object is an instance of a class that can contain data (in the form of variables or properties) and code (in the form of methods).
- **Structures (structs):** A data type that groups several elements, which can be of different types. Typically used in structured programming languages.

2.2.4. Special Data Types

- **Null:** Represents the absence of a value.
- **Pointers:** Used to index and store the memory address of another variable (common in languages like C or C++).

2.3. Characteristics of Data Types

The characteristics of data types in programming languages (including MATLAB), such as integers (int), floating-point numbers (float and double), characters (char), and Booleans (bool), are presented below:

2.3.1. Integers (int)

- **Description:** Represent whole numbers, i.e., numbers without a decimal part.
- **Characteristics:**
  - **Size:** The size of an integer can vary depending on the language and system architecture (typically 16, 32, or 64 bits).
  - **Range of values:** Depends on the size of the integer. For example, a 32-bit signed integer can range from -2,147,483,648 to 2,147,483,647.
  - **Signed/Unsigned:** Integers can be signed (can be positive or negative) or unsigned (always positive).
  - **Operations:** Support standard arithmetic operations such as addition, subtraction, multiplication, and division.

2.3.2. Floating-Point Numbers (float, double)

- **Description:** Represents real numbers, i.e., numbers that can have a decimal part.
- **Characteristics:**
  - **Precision:**
    - **float:** Typically uses 32 bits to represent a floating-point number. Offers a precision of about 7 significant digits.
    - **double:** Typically uses 64 bits, offering a precision of about 15 significant digits.

- **Range of Values:** Varies according to size. For example, a float can represent values on the order of ±3.4E38, while a double can go up to ±1.7E308.
- **Precision:** Floating-point numbers are approximate, which can lead to rounding errors during calculations.
- **Usage:** Floats are used when memory is limited and lower precision is acceptable. Doubles are preferred for calculations requiring higher precision.
- **Operations:** Operations include addition, subtraction, multiplication, division, and advanced mathematical functions like sine, cosine, etc.

2.3.3. Characters (char)

- **Description:** Represents a single character (letter, number, and symbol).
- **Characteristics:**
  - **Size:** Typically 8 bits, capable of representing 256 distinct values (0-255), according to the ASCII table.
  - **Representation:** Characters are often represented by their ASCII code, for example, 'A' corresponds to 65.
  - **Usage:** Primarily for manipulating individual letters, numbers, or symbols in character strings.
  - **Encoding:** Characters can be extended to 16 bits (or more) to support larger character sets like Unicode.

2.3.4. Booleans (bool)

- **Description:** Represents truth values, i.e., true or false.
- **Characteristics:**
  - **Size:** Often implemented with a single bit, but in many systems, a Boolean can occupy a full byte (8 bits).
  - **Possible Values:** true or false.
  - **Usage:** Essential for logical operations, conditions, and control flow (such as in if, while, for structures).
  - **Operations:** Include logical operations such as AND (&&), OR (||), NOT (!).

2.3.5. Examples in C++ (cpp)

For a better understanding, use this code in MATLAB.

1. **Int:** int integer = 42; // 32-bit integer
2. **Float:** float reel_float = 3.14f; // Single precision floating-point number (32 bits)
3. **Double:** double reel_double = 3.14159; // Double precision floating-point number (64 bits)
4. **Char:** char caractere = 'A'; // Character, uses 8 bits
5. **Bool:** bool boolean = true; // Boolean, truth value

2.3.6. MATLAB Code Example

For a better understanding, use this code in MATLAB.

```
% Integer (int)
entier = int32(42); % Creates a 32-bit integer
disp(['Integer: ', num2str(entier)]);
% Floating-point number (float)
reel_float = single(3.14); % Creates a single precision floating-point number (32 bits)
disp(['Float: ', num2str(reel_float)]);
% Floating-point number (double)
reel_double = 3.14159; % By default, MATLAB uses double precision (64 bits)
disp(['Double: ', num2str(reel_double)]);
% Character (char)
caractere = 'A'; % Creates a character
disp(['Character: ', caractere]);
% Boolean (bool)
boolean = true; % Creates a Boolean value
disp(['Boolean: ', num2str(boolean)]);
% Comparing the integer with another value
if entier > 20
disp('The integer is greater than 20.');
else
disp('The integer is less than or equal to 20.');
end
% Operation on floating-point numbers
somme = reel_float + reel_double;
disp(['Sum of floating-point numbers: ', num2str(somme)]);
% Concatenation of characters
chaine = [caractere, 'BC'];
disp(['Character string: ', chaine]);
```

2.4. MATLAB Variables

Variables in MATLAB are containers that allow you to store and manipulate data. MATLAB is a programming language widely used for numerical calculations, and it automatically manages the creation and handling of variables. Here are the key concepts related to variables in MATLAB:

2.4.1. Creating Variables

A variable is created simply by assigning it a value with the assignment operator (=), for example:

Use this code in MATLAB for better understanding:

```
x = 10; % Creates a variable x and assigns it the value 10
y = 3.14; % Creates a variable y and assigns it the value 3.14
nom = 'Alice'; % Creates a variable nom that contains the string 'Alice'
```

2.4.2. Types of Variables

MATLAB is a dynamically typed language, meaning you don't need to explicitly declare the type of a variable. The type is automatically determined based on the assigned value.

- **Common Types:**
  - **Scalars:** A single integer or real number, such as 5 or 3.14.
  - **Complex:** Complex numbers can be created by adding an "i" suffix to a number (e.g., 3 + 4i).
  - **Logical Variables:** Boolean variables can take the values true (1) or false (0).
  - **Character Strings:** Sequence of characters (char), such as 'Hello' or "Hi" (MATLAB now supports both char and string types).
  - **Vectors and Matrices:** Vectors and matrices can contain real, complex, Boolean, or even character string values.
    - **Vectors:** A vector is a one-dimensional array, which can be a row or a column.
    - **Matrices:** A matrix is a two-dimensional array. MATLAB is designed to perform matrix operations easily, making it a powerful tool for matrix computations.
  - **Cells:** Flexible containers that can hold different data types, like C = {1, "text", [1,2,3]}.
  - **Structures:** Containers for grouping related data, accessible via fields, like person.name = 'Alice'; person.age = 25;.
  - **Object Arrays:** Object-oriented programming, where classes define objects with properties and methods.

2.4.3. Variable Names

Variable names must start with a letter, followed by letters, numbers, or underscores (_). MATLAB is case-sensitive, meaning Nom and nom are two different variables.

For example, use this code in MATLAB for better understanding:

```
1. temperature = 25;
2. Temperature = 30; % Different from 'temperature'
```

2.4.4. Operations on Variables

You can perform mathematical, logical, or textual operations on variables.

Examples: Use this code in MATLAB Editor for better understanding:

```
1. a = 5;
2. b = 10;
3. somme = a + b;  % Addition
4. produit = a * b;  % Multiplication
5. nomComplet = [nom, ' Bob'];  % Concatenation of character strings
```

2.4.5. Displaying and Deleting Variables

2.4.5.1. Displaying

Variables can be displayed in the command window by simply typing their name or using the ***disp()*** function:

- Example in MATLAB:

```
1. disp(x);  % Displays the value of x
```

2.4.5.2. Deleting

To delete a variable from memory, use the clear command:

- Example in MATLAB:

```
1. clear x;  % Deletes the variable x
2. clear;    % Deletes all variables
```

2.4.6. Saving and Loading Variables

You can save variables in a MAT file and reload them later.

2.4.6.1. Saving

- Example in MATLAB:

```
1. save myFile.mat x y;  % Saves the variables x and y in a file
```

2.4.6.2. Loading

- Example in MATLAB:

```
1. load myFile.mat;  % Loads all variables from the file
```

2.4.7. Workspaces

MATLAB manages a workspace where all variables created during a session are stored. You can view all current variables with the ***who*** command or ***whos*** for additional details.

- Example in MATLAB:

```
1. who;   % Lists variables in the workspace
2. whos;  % Lists variables with details about their type and size
```

2.4.8. Multiple Assignment

MATLAB allows assigning multiple variables at once:

- Example in MATLAB:

```
1. [a, b, c] = deal(1, 2, 3);  % Assigns 1 to a, 2 to b, and 3 to c
```

2.4.9. Global and Local Variables

In MATLAB, variables can be either local to a function or global across multiple functions or the global workspace. Below is a detailed explanation of global and local variables, along with examples of usage.

2.4.9.1. Local Variables

A local variable is a variable that is defined and used within a function. It is only accessible inside that function and is not visible or accessible outside of it.

- **Characteristics:**
  - Each function in MATLAB has its own local workspace, distinct from the global workspace.
  - Variables defined inside a function are destroyed when the function ends, unless they are returned as outputs.
- Example: In this example, a, b, and c are all local variables within the SUM function. They are not accessible outside of this function.

```
function result = SUM(a, b)
% a and b are local variables
c = a + b;  % 'c' is also a local variable
result = c;
end
```

2.4.9.2. Global Variables

A global variable is a variable that can be shared between multiple functions or the global MATLAB workspace. It is defined using the global keyword.

- **Characteristics:**
  - Global variables are accessible from any function that declares them with global.
  - They persist in the workspace until explicitly cleared or MATLAB is closed.
  - Global variables must be declared as global in each function where they are used.
- Example: In this example, x is a global variable. It is initialized in the main workspace, then modified inside the increaseX function.

```
% Declaring a global variable
global x;
x = 10;
```

```
% Function that uses the global variable
function increaseX()
global x;  % Redeclaration of x as global
x = x + 1;
end
```

```
% Calling the function
increaseX();
disp(x);  % Displays 11
```

2.4.9.3. Key Differences between Local and Global Variables

- **Scope:**
  - **Local:** Visible only within the function where it is defined.
  - **Global:** Accessible in any script or function that declares it as global.
- **Persistence:**
  - **Local:** The variable is destroyed when the function ends (unless returned as a result).
  - **Global:** Persists in the global workspace until explicitly cleared or MATLAB is closed.
- **Usage:**
  - **Local:** Used to encapsulate data and calculations specific to a function without interfering with the rest of the program.
  - **Global:** Used when multiple functions need to share and modify the same data.

2.4.9.4. Considerations

- **Risks of Global Variables:**
  - They can lead to name conflicts if multiple functions use global variables with the same name.
  - They make debugging more difficult since any change in one function can affect other parts of the program.
- **Best Practices:**
  - Limit the use of global variables and prefer local variables and parameter passing between functions for better modularity and code maintenance.

2.4.10. Complete Example

In this example:

- **Counter**: is a global variable, modified and displayed by the functions incrementCounter and displayCounter.
- **Sum**: is a local variable, defined and used only within the function calculateSum. It is not accessible outside of this function.

Main Script

```
1. global counter;
2. counter = 0;

3. function incrementCounter()
4. global counter;
5. counter = counter + 1;
6. end

7. function displayCounter()
8. global counter;
9. disp(['Counter value: ', num2str(counter)]);
10. end
```

```
% Using the functions
incrementCounter();
displayCounter();  % Displays 1

incrementCounter();
displayCounter();  % Displays 2

% Local variables (not accessible outside their function)
function calculateSum(a, b)
sum = a + b;  % sum is local to this function
disp(['Sum: ', num2str(sum)]);
end

% Calling the function with local arguments
calculateSum(5, 7);  % Displays 12
```

Note:

- The variable sum is not accessible here because it is local to ***calculateSum()***.
- The command ***disp(sum)***; would result in an error.

2.5. Complex Number Variables

Complex numbers are widely used in fields such as:

- **Signal Processing**: For analyzing signals that can be represented as complex functions.
- **Systems Theory**: In the analysis of the stability of dynamic systems.
- **Electromagnetism**: To represent electric and magnetic fields.

In mathematics, complex numbers are variables that contain a real part and an imaginary part. MATLAB natively handles complex numbers, making complex calculations straightforward.

2.5.1. Creating Complex Numbers

A complex number can be created by combining a real part and an imaginary part. In MATLAB, the imaginary part is indicated with i.

Example: In these examples, z1 is the complex number 3+4i and z2 is the complex number 2−5i.

```
z1 = 3 + 4i;  % Complex number with real part 3 and imaginary part 4
z2 = 2 - 5i;  % Complex number with real part 2 and imaginary part -5
```

2.5.2. Operations on Complex Numbers

2.5.2.1. Addition and Subtraction

```
z1 = 1 + 2i;
z2 = 2 - 3i;
```

3. *sum = z1 + z2;* *% Sum of the two complex numbers*
4. *difference = z1 - z2;* *% Difference of the two complex numbers*

2.5.2.2. Multiplication and Division

1. *product = z1 * z2;* *% Product of z1 and z2*
2. *quotient = z1 / z2;* *% Division of z1 by z2*

2.5.2.3. Conjugate

The conjugate of a complex number reverses the sign of the imaginary part.

1. *conjZ = conj(z1);* *% Conjugate of z1*

2.5.2.4. Magnitude and Argument

The magnitude (or modulus) of a complex number is its distance from the origin in the complex plane, while the argument is the angle with the real axis.

2. *magnitudeZ = abs(z1);* *% Magnitude of z1*
3. *argumentZ = angle(z1);* *% Argument (angle in radians) of z1*

2.5.3. Mathematical Functions with Complex Numbers

MATLAB supports many mathematical functions for complex numbers, including exponentiation, trigonometric functions, and logarithms.

Example:

1. *z = 1 + 2i;*
2. *exponential = exp(z);* *% Exponential of z*
3. *sine = sin(z);* *% Sine of z*
4. *logarithm = log(z);* *% Natural logarithm of z*
5. *power = z^2;* *% z raised to the power of 2*

2.5.4. Conversion between Cartesian and Polar Representations

- **Cartesian Representation**: A complex number is expressed as z=a+bi.
- **Polar Representation**: A complex number is expressed in terms of magnitude r and argument θ: $z = r \times e^{i\theta}$.

1. *% To convert from Cartesian to polar form*
2. *r = abs(z);* *% Magnitude (r)*
3. *theta = angle(z);* *% Argument (θ)*

4. *% To create a complex number from its polar form*
5. *z = r * exp(1i * theta);* *% Polar to Cartesian conversion*

2.5.5. Complete Example: Complex Numbers

1. *% Creating complex numbers*
2. *z1 = 3 + 4i;*
3. *z2 = 1 - 2j;*

```
% Basic operations
sum = z1 + z2;
difference = z1 - z2;
product = z1 * z2;
quotient = z1 / z2;

% Conjugate, magnitude, and argument
conjugateZ1 = conj(z1);
magnitudeZ1 = abs(z1);
argumentZ1 = angle(z1);

% Mathematical functions
expZ1 = exp(z1);
sinZ1 = sin(z1);
logZ1 = log(z1);

% Cartesian to polar conversion and back
r = abs(z1);
theta = angle(z1);
z_polar = r * exp(1i * theta);

% Displaying results
disp(['Sum: ', num2str(sum)]);
disp(['Difference: ', num2str(difference)]);
disp(['Product: ', num2str(product)]);
disp(['Quotient: ', num2str(quotient)]);
disp(['Conjugate of z1: ', num2str(conjugateZ1)]);
disp(['Magnitude of z1: ', num2str(magnitudeZ1)]);
disp(['Argument of z1: ', num2str(argumentZ1)]);
disp(['Exponential of z1: ', num2str(expZ1)]);
disp(['Sine of z1: ', num2str(sinZ1)]);
disp(['Logarithm of z1: ', num2str(logZ1)]);
disp(['z1 in polar form: ', num2str(z_polar)]);
```

2.6. Boolean Variables

Boolean variables are essential for:

- **Program flow control** with conditions (if, switch) and loops (while, for).
- **Data analysis**, for example, filtering values in arrays using logical masks.
- **Condition checking** in algorithms, such as convergence criteria or stopping conditions.

MATLAB offers robust support for logical operations, allowing for the construction of flexible and robust scripts and functions.

In MATLAB, ***boolean*** (or ***logical***) variables are used to represent truth values, i.e., true or false. These values are particularly useful for conditions, loops, and logical operations.

2.6.1. Creating Boolean Variables

A boolean variable can be created using the values true and false.

1. *a = true;* *% a is a boolean variable containing the value true*
2. *b = false;* *% b is a boolean variable containing the value false*
3. *MATLAB also treats 1 as true and 0 as false.*
4. *c = (5 > 3);* *% c is true because 5 is greater than 3*
5. *d = (2 == 3);* *% d is false because 2 is not equal to 3*

2.6.2. Logical Operations

Boolean variables can be manipulated with logical operations such as AND, OR, NOT, and XOR.

2.6.2.1. Logical AND

Returns true if both operands are true.

1. *x = true;*
2. *y = false;*
3. *z = x && y;* *% z is false because y is false*

2.6.2.2. Logical OR

Returns true if at least one of the operands is true.

1. *z = x || y;* *% z is true because x is true*

2.6.2.3. Logical NOT

Inverts the logical value.

2. *z = ~x;* *% z is false because x is true*

2.6.2.4. Logical XOR

Returns true if only one of the operands is true, but not both.

3. *z = xor(x, y);* *% z is true because only one (x or y) is true*

2.6.3. Use in Conditions and Loops

Boolean variables are often used to control program flow through conditional structures and loops.

2.6.3.1. Conditional if

4. *if a*
5. *disp('a is true');*
6. *else*
7. *disp('a is false');*
8. *end*

2.6.3.2. while Loops

```
1. counter = 0;
2. continueLoop = true;
3. while continueLoop
4. counter = counter + 1;
5. if counter >= 5
6. continueLoop = false;
7. end
8. end
9. disp(['Counter reached ', num2str(counter)]);
```

2.6.4. Relational Operators

Boolean variables are often the result of relational operators, which compare two values.

2.6.4.1. Equality (==)

```
1. a = (5 == 5);  % a is true because 5 equals 5
```

2.6.4.2. Inequality (~=)

```
2. b = (5 ~= 3);  % b is true because 5 is not equal to 3
```

2.6.4.3. Greater Than (>) and Greater Than or Equal (>=)

```
3. c = (7 > 3);   % c is true because 7 is greater than 3
4. d = (5 >= 5);  % d is true because 5 is greater than or equal to 5
```

2.6.4.4. Less Than (<) and Less Than or Equal (<=)

```
1. e = (2 < 3);   % e is true because 2 is less than 3
2. f = (4 <= 4);  % f is true because 4 is less than or equal to 4
```

2.6.5. Logical Arrays (see Section 2.8.)

Boolean variables can also be stored in logical arrays, where each element of the array is either true or false.

1. **Creating Logical Arrays**

```
2. logical_array = [true, false, true; false, true, false];
```

2.6.5.1. Operations on Logical Arrays

Logical operations can be applied element-wise to arrays.

1. **Example**:

```
2. a = [true, false, true];
3. b = [false, true, true];

4. c = a & b;  % c = [false, false, true]
5. d = a | b;  % d = [true, true, true]
```

2.6.6. Complete Example

- **Creating Boolean Variables**

```
a = true;
b = false;
```

- **Logical Operations**

```
logical_and = a && b;     % false
logical_or = a || b;      % true
logical_not = ~a;         % false
logical_xor = xor(a, b);  % true
```

- **Using in a Condition**

```
if logical_and
disp('Both values are true.');
else
disp('At least one value is false.');
end
```

- **Logical Array**

```
array = [true, false, true; false, true, false];
disp('Logical array:');
disp(array);
```

- **Array Operation**

```
result = array & [true, true, false; false, false, true];
disp('Result of logical AND on the array:');
disp(result);
```

2.7. Strings

Strings are sequences of characters used to represent text. MATLAB offers various methods for creating, manipulating, and analyzing strings. Strings are used in various contexts, such as:

- **Text Manipulation**: Processing and analyzing textual data.
- **User Interface**: Displaying messages and instructions.
- **Report Generation**: Dynamically creating formatted text and results.

2.7.1. Creating Strings

Strings: Strings are created by enclosing text in single quotes (') or double quotes (").

- **Example**:

```
str1 = 'Hello, MATLAB!';  % Using single quotes
str2 = "Hello, MATLAB!";  % Using double quotes
```

Empty Strings: An empty string can be created using two single quotes with no space between them ('') or two double quotes ("").

- **Example**:

```
empty_str1 = '';  % Empty string with single quotes
empty_str2 = "";  % Empty string with double quotes
```

2.7.2. Basic String Operations

Concatenation: Strings can be combined using the [ ] operator or the ***strcat*** function.

- **Example**:

```
part1 = "Hello";
part2 = " world!";
fullStr = [part1, part2];       % Concatenation using brackets
fullStr2 = strcat(part1, part2); % Concatenation using strcat
```

String Length: The strlength or length function returns the number of characters in a string.

- **Example**:

```
len = strlength(fullStr);  % Length of the string
```

Accessing String Elements: Individual characters can be accessed via their indices.

- **Example**:

```
firstChar = fullStr(1);    % First character of the string
lastChar = fullStr(end);   % Last character of the string
```

2.7.3. String Manipulation

2.7.3.1. Splitting Strings

Extracting Substrings: The extractBetween function extracts a substring between two indices.

- **Example**:

```
subStr = extractBetween(fullStr, 1, 5);  % Extracts "Hello"
```

Splitting Strings: The split function divides a string into multiple parts using a delimiter.

- **Example**:

```
parts = split("One-two-three", "-");  % Splits the string into ["One", "two", "three"]
```

2.7.3.2. Modifying Strings

Replacing Substrings: The replace function replaces part of the string with another text.

- **Example**:

```
strModified = replace(fullStr, "Hello", "Hi");  % Replaces "Hello" with "Hi"
```

Trimming Whitespace: The strtrim function removes leading and trailing whitespace.

- **Example**:

```
trimmedStr = strtrim("  MATLAB  is  awesome!  ");% Removes spaces
```

2.7.3.3. Comparing Strings

String Comparison: The strcmp function compares two strings for equality.

- **Example**:

```
isEqual = strcmp("MATLAB", "MATLAB");  % Returns true
```

Case-Insensitive Comparison: The strcmpi function compares two strings ignoring case.

- **Example**:

```
isEqualIgnoreCase = strcmpi("matlab", "MATLAB");  % Returns true
```

2.7.3.4. Searching in Strings

Checking for Substrings: The contains function checks if a string contains a specific substring.

- **Example**:

```
containsHello = contains(fullStr, "Hello");  % Returns true if "Hello" is in fullStr
```

Checking Start and End of Strings: The ***startsWith*** and ***endsWith*** functions check if a string starts or ends with a specific substring.

- **Example**:

```
startsWithHello = startsWith(fullStr, "Hello");  % Returns true if fullStr starts with "Hello"
endsWithWorld = endsWith(fullStr, "world!");  % Returns true if fullStr ends with "world!"
```

2.7.3.5. Converting Between Strings and Numbers

Converting Numbers to Strings: The num2str function converts a number to a string.

- **Example**:

```
num = 123;
strNum = num2str(num);  % "123"
```

Converting Strings to Numbers: The str2num or str2double function converts a string to a number.

- **Example**:

```
str = "456";
numVal = str2double(str);  % 456 as a number
```

2.8. Vectors

Vectors are one of the most commonly used data structures. They are essentially one-dimensional arrays that can hold numbers, strings, or other data types. Vectors can be classified into two types: row vectors (horizontal) and column vectors (vertical).

Vectors are used in various fields, such as:

- **Scientific and Engineering Calculations:** Representation of signals, time series, or experimental data.
- **Mathematical Modeling:** Utilization in linear systems of equations, optimizations, etc.
- **Image and Sound Processing:** Manipulation of pixels or audio samples.

2.8.1. Creating Vectors

Row Vector: A row vector is created by listing elements between brackets ([ ]), separated by spaces or commas.

```
row_vector = [1 2 3 4 5];  % or [1, 2, 3, 4, 5]
```

Column Vector: A column vector is created by listing elements between brackets, separated by semicolons (;), or by transposing a row vector using the apostrophe (').

```
col_vector = [1; 2; 3; 4; 5];  % Or row_vector' to transpose
```

2.8.2. Accessing Vector Elements

Elements of a vector can be accessed using indices. MATLAB uses 1-based indexing (unlike many other programming languages, such as C++, where indexing starts at 0).

2.8.2.1. Accessing a Specific Element

1. *first = row_vector(1); % First element of the row vector*
2. *last = col_vector(end); % Last element of the column vector*

2.8.2.2. Accessing Multiple Elements

3. *sub_vector = row_vector(2:4); % Elements 2 to 4 of the row vector*

2.8.2.3. Vector Operations

Addition and Subtraction: Arithmetic operations can be performed element by element between vectors of the same size.

1. *vec1 = [1 2 3];*
2. *vec2 = [4 5 6];*
3. *sum = vec1 + vec2; % [5 7 9]*
4. *difference = vec1 - vec2; % [-3 -3 -3]*
5. **Element-wise Multiplication and Division:** Element-wise multiplication and division are *performed using .* and ./.*
6. *product = vec1 .* vec2; % [4 10 18]*
7. *division = vec2 ./ vec1; % [4 2.5 2]*

Dot Product: The dot product of two vectors is calculated using the * operator.

8. *dot_product = vec1 * vec2'; % 32 (1*4 + 2*5 + 3*6)*

2.8.2.4. Useful Functions for Vectors

Vector Length: The length function returns the number of elements in a vector.

9. *n = length(row_vector); % 5*

Sum of Elements: The sum function calculates the sum of vector elements.

10. *sum_elements = sum(row_vector); % 15*

Product of Elements: The prod function calculates the product of all elements.

11. *product_elements = prod(col_vector); % 120 (1*2*3*4*5)*

Minimum and Maximum: The min and max functions return the smallest and largest element, respectively.

12. *min_val = min(row_vector); % 1*
13. *max_val = max(row_vector); % 5*

Mean and Standard Deviation: The mean and ***std*** functions calculate the mean and standard deviation.

14. *mean_val = mean(row_vector); % 3*
15. *std_val = std(row_vector); % 1.5811*

2.8.2.5. Generating Special Vectors

Regular (Linear) Vectors: The ***linspace*** function generates a vector of values evenly spaced between two bounds.

1. *lin_vec = linspace(0, 10, 5); % [0 2.5 5 7.5 10]*

Equidistant Value Vectors: The colon operator (***:***) is used to generate sequences with a fixed step.

2. *colon_vec = 0:2:10; % [0 2 4 6 8 10]*

Zero or One Vectors: The ***zeros*** *and* ***ones*** functions create vectors filled with zeros or ones.

3. *zero_vec = zeros(1, 5); % [0 0 0 0 0]*
4. *one_vec = ones(1, 5); % [1 1 1 1 1]*

Random Vectors: The ***rand and randi*** functions generate vectors of random numbers.

5. *rand_vec = rand(1, 5); % Vector of 5 random uniform numbers between 0 and 1*
6. *rand_int_vec = randi(10, 1, 5); % Vector of 5 random integers between 1 and 10*

2.9. Polynomials

Polynomials are represented by vectors that contain the coefficients of the polynomial terms, from the highest degree to the lowest. MATLAB provides various functions for manipulating, evaluating, differentiating, integrating, and finding the roots of polynomials.

Polynomials are ubiquitous in scientific and technical fields, particularly for:

- **Mathematical Modeling:** Approximations of complex functions.
- **Signal Processing:** Filtering and analyzing signals.
- **Control and Automation:** Designing control systems.
- **Numerical Computation:** Solving algebraic equations.

2.9.1. Polynomial Representation

In mathematics, a polynomial of degree n is of the form:

$$P(x) = a_n x^n + \cdots + a_1 x + a_0$$

In MATLAB, this polynomial is represented by a vector of coefficients:

1. *p = [a_n, a_{n-1}, \dots, a_1, a_0];*

Example: The polynomial $p(x) = 2x^3 + 3x^2 + 4x + 5$ is represented by:

2. *p = [2 3 4 5];*

2.9.2. Polynomial Evaluation

To evaluate a polynomial at a given value of x, the ***polyval*** function is used.

3. *x = 2;*
4. *value = polyval(p, x); % Evaluates the polynomial p(x) = 2x^3 + 3x^2 + 4x + 5 at x = 2*

2.9.3. Polynomial Operations

2.9.3.1. Addition and Subtraction

Polynomials can be added or subtracted by adding or subtracting the coefficient vectors.

1. *p1 = [1 2 3]; % Represents the polynomial x^2 + 2x + 3*
2. *p2 = [4 5]; % Represents the polynomial 4x + 5*

% Addition

3. *p3 = p1 + [0 p2]; % p3 represents x^2 + 6x + 8*

% Subtraction

4. *p4 = p1 - [0 p2]; % p4 represents x^2 - 2x - 2*

2.9.3.2 Multiplication

Polynomials can be multiplied using the ***conv*** function, which performs the convolution of the coefficients.

5. *p1 = [1 2 3]; % Represents the polynomial x^2 + 2x + 3*
6. *p2 = [4 5]; % Represents the polynomial 4x + 5*
7. *p3 = conv(p1, p2); % p3 represents 4x^3 + 13x^2 + 22x + 15*

2.9.4. Polynomial Division

To divide one polynomial by another, the deconv function is used, which returns the quotient and remainder.

8. *p1 = [1 -3 2]; % Represents x^2 - 3x + 2*
9. *p2 = [1 -1]; % Represents x – 1*
10. *[quotient, remainder] = deconv(p1, p2);*
11. *% quotient represents x – 2*
12. *% remainder represents 0*

2.9.5. Polynomial Roots

The roots of the polynomial (the values of xxx for which p(x)=0p(x) = 0p(x)=0) can be found using the roots function.

13. *roots = roots(p); % Finds the roots of the polynomial p(x) = 2x^3 + 3x^2 + 4x + 5*

2.9.6. Differentiation and Integration

2.9.6.1 Differentiation

The derivative of a polynomial is calculated by multiplying each coefficient by its exponent and reducing the exponent by 1 using the polyder function.

14. *p_prime = polyder(p); % Calculates the derivative of p(x)*

2.9.6.2 Integration

The integral of a polynomial is obtained by adding an exponent to each term and dividing the coefficient by the new exponent. MATLAB allows you to integrate a polynomial using the polyint function.

15. *p_int = polyint(p); % Calculates the integral of p(x)*

2.9.7. Dot Product

To evaluate the dot product of two polynomials, you can use polynomial multiplication with the conv function.

16. *p1 = [1 2 3]; % Represents the polynomial x^2 + 2x + 3*
17. *p2 = [4 5]; % Represents the polynomial 4x + 5*
18. *dot_product = conv(p1, p2); % p(x) = 4x^3 + 13x^2 + 22x + 15*

2.10. Matrices

Matrices are one of the most fundamental and powerful data structures in MATLAB. A matrix is a two-dimensional rectangular array of numbers arranged in rows and columns, which can contain real, complex, or even other types of data. Matrix operations are central to numerous scientific and engineering applications, including linear algebra, signal processing, and system modeling. MATLAB, short for "MATrix LABoratory," is specifically designed to manipulate and perform efficient calculations on matrices.

Matrices are ubiquitous in many fields:

- **Scientific and Engineering Computation:** Physical modeling, solving systems of equations, geometric transformations.
- **Image Processing:** Pixel manipulation and image filtering.
- **Economics and Finance:** Data analysis, modeling, and forecasting.
- **Statistics:** Analysis of variance, linear regression.

2.10.1. Creating Matrices

Matrices in MATLAB are created by enclosing elements within brackets [ ]. Elements within the same row are separated by spaces or commas, and rows are separated by semicolons ";".

1. *A = [1 2 3; 4 5 6; 7 8 9];*

This creates a matrix named "A" of dimension 3x3: $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

2.10.2. Accessing Matrix Elements

You can access matrix elements using row and column indices.

Accessing a Specific Element:

2. *element = A(2, 3); % Accesses the element in the 2nd row and 3rd column (in this case, 6)*

Accessing an Entire Row or Column:

3. *row = A(2, :); % Accesses the 2nd row (here, [4 5 6])*
4. *column = A(:, 3); % Accesses the 3rd column (here, [3; 6; 9])*

2.10.3. Matrix Operations

2.10.3.1. Addition and Subtraction

Matrices of the same size can be added or subtracted element by element.

5. *B = [9 8 7; 6 5 4; 3 2 1];*
6. *C = A + B; % Addition of A and B*
7. *D = A - B; % Subtraction of B from A*

2.10.3.2. Multiplication

There are two types of multiplication on matrices:

Matrix Multiplication:

```
8.  E = A * B;  % Matrix multiplication (matrix product)
```

Element-wise Multiplication (also called Hadamard product):

```
9.  F = A .* B;  % Element-wise multiplication
```

2.10.3.3. Transposition

The transpose of a matrix is obtained using the apostrophe (').

```
10. G = A';  % Transpose of A
```

2.10.3.4. Inverse and Determinant

Matrix Inverse: Calculated with inv (works only for non-singular square matrices).

```
11. H = inv(A);  % Inverse of A
```

Matrix Determinant: Calculated with det.

```
12. det_A = det(A);  % Determinant of A
```

2.10.4. Matrix Manipulation

MATLAB offers numerous functions for manipulating matrices.

2.10.4.1. Reshaping

To change the size: the reshape function modifies the matrix's dimensions without changing its data.

```
13. I = reshape(A, [1, 9]);  % Reshapes A into a 1x9 row vector
```

2.10.4.2. Concatenation

Matrices can be concatenated horizontally or vertically.

Horizontal Concatenation:

```
14. J = [A B];  % Concatenates A and B horizontally
```

Vertical Concatenation:

```
15. K = [A; B];  % Concatenates A and B vertically
```

2.10.4.3. Special Matrices

MATLAB provides functions to quickly create special matrices:

Identity Matrix: eye(n) creates an identity matrix of size n×n.

```
16. L = eye(3);  % 3x3 Identity matrix
```

Zero Matrix: zeros(m, n) creates an m-by-n matrix of zeros.

```
17. M = zeros(3, 2);  % 3x2 Matrix of zeros
```

One Matrix: ones(m, n) creates an m-by-n matrix of ones.

```
18. N = ones(2, 3);  % 2x3 Matrix of ones
```

Random Matrix: rand(m, n) creates an m×n matrix of uniformly distributed random numbers between 0 and 1.

```
19. = rand(2, 2);  % 2x2 Matrix of random numbers
```

2.10.5. Solving Linear Systems

MATLAB is particularly efficient for solving systems of linear equations. If you have a linear system AX=B, where A is a square matrix and B is a vector, you can directly find X using:

```
20. X = A \ B; % Solves AX = B
```

Part 02: Simulation Part (Matrix Operations)

To continue with the second part of the tutorial, you need to create a new folder named "TP04_INFO3" within the "TP_INFO3" directory and set it as the current folder in MATLAB.

In this part, you'll explore various operations on matrices and vectors. MATLAB provides a wide range of operations, including addition, multiplication, transposition, and element-wise operations. For element-wise operations, you typically prefix the operator with a period ('.').

You will work with two matrices, A and B, as follows:

$$A = \begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix},\ A1 = \begin{pmatrix} 4 & 1 & 2 \\ 6 & 1 & 6 \\ 3 & 3 & 1 \end{pmatrix},\ B = \begin{pmatrix} 8 & 1 \\ 5 & 7 \end{pmatrix},\ V_1 = \begin{pmatrix} 5 \\ 4 \\ 6 \end{pmatrix},$$

$$V_2 = (2 \quad 1 \quad 6), V_3 = (4 \quad 9),$$

You can perform different matrix operations using these matrices. If you have any specific questions or tasks related to matrix operations, please let me know, and I'll be happy to assist you further.

Exercise 1 (Concatenate, Compare matrices)

Using Matlab (Command Window space);

1. Matrix C includes matrix A and A1 horizontally: $C = [A, A1]$
2. Use the function $cat(dim, A, A1)$ to assemble the matrices A and A1:
$$C1 = cat(2, A, A1)$$
3. Compare the matrix C and C1, use the function: $isequal(C, C1)$
4. Compare matrix A and A1, use: $A == A1$
5. Conclude!!

Exercise 2: Concatenate and Compare Matrices

In this exercise, you will practice concatenating and comparing matrices using MATLAB.

1. Create a new script in the "TP04_INFO3" folder and save it with the name "ConcatCompare_TP4.m".
2. Define two matrices, C and D, as follows:
$$C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}; D = \begin{pmatrix} 50 & 60 \\ 70 & 80 \end{pmatrix};$$
3. Concatenate the matrices C and D horizontally and assign the result to a new matrix E.
$$E = [C, D];$$

4. Compare the matrices C and D element-wise and assign the result to a logical matrix F.

$$F = (C == D);$$

5. Display the matrices E and F using the "*disp*" function.

disp('Matrix E:');
disp(E);
disp('Matrix F (Element-wise comparison result):');
disp(F);

6. Run the script and observe the concatenated matrix E and the element-wise comparison result matrix F.

Feel free to modify and extend this exercise based on your learning needs. If you have any specific questions or tasks related to the exercise, please let me know, and I'll be here to assist you further.

Exercise 3 (Add – Delete (row/column) in a matrix)

Using Matlab (Command Window space);

1. Calculate the dimension of the matrix A and B.
2. Calculate the number of elements in the vector V1 and V2.
3. Add the vector V1 in the 3rd column of matrix B.

Table 1. Add – Delete (row/column) in a matrix

Method 1	**Method 2**

`>> B(1,3)=V1(1);`
`>> B(2,3)=V1(2);`
`>> B(3,3)=V1(3);`
`>> B`
 	
`B =`
 	
`8 1 5`
`5 7 4`
`0 0 6`
	`>> B=[8 1; 5 7];`
	`>> B(3,1)=0;`
	`>> B=[B, V1]`
	
	`B =`
	
	`8 1 5`
	`5 7 4`
	`0 0 6`

Here's how you can perform the tasks using MATLAB in the Command Window:

1. *% Calculate dimensions*
2. *dim_A = size(A);*
3. *dim_B = size(B);*
4. *% Calculate number of elements in vectors*
5. *num_elements_V1 = numel(V1);*
6. *num_elements_V2 = numel(V2);*
7. *% Add vector V1 as a new column in matrix B*
8. *B_with_V1 = [B V1];*
9. *% Display results*
10. *disp("Dimension of matrix A:");*
11. *disp(dim_A);*
12. *disp("Dimension of matrix B:");*

```
13. disp(dim_B);
14. disp("Number of elements in vector V1:");
15. disp(num_elements_V1);
16. disp("Number of elements in vector V2:");
17. disp(num_elements_V2);
18. disp("Matrix B with vector V1 added as a new column:");
19. disp(B_with_V1);
```

Copy and paste the above code into the MATLAB Command Window to execute the tasks. It will display the dimensions of matrices A and B, the number of elements in vectors V1 and V2, and the matrix B with vector V1 added as a new column.

```
>> B(3,:)=V2

B =

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

Figure 1. Add vector V2 in the 3rd row of matrix B.

Here's how you can add vector V2 as a new row in the 3rd row of matrix B using MATLAB:

```
1. % Given matrices and vector
2. B = [8 1; 5 7];
3. V2 = [2 1 6];
4. % Add vector V2 as a new row in the 3rd row of matrix B
5. B_with_V2 = [B; V2];
6. % Display matrix B with vector V2 added as a new row
7. disp("Matrix B with vector V2 added as a new row:");
8. disp(B_with_V2);
```

Copy and paste the above code into the MATLAB Command Window to execute the task. It will display the matrix B with vector V2 added as a new row in the 3rd row.

Table 2. Delete the second row of matrix A and assign the new matrix to C3

Method 1	**Method 2**

`>> C1=[A(1,:)];`
`>> C2=[A(3,:)];`
`>> C3=[C1; C2]`
 	
`C3 =`
 	
`8 1 6`
`4 9 2`
	`>> A(2,:)=[]`
	
	`A =`
	
	`8 1 6`
	`4 9 2`
	
	`>> C3=A;`

Exercise 4 (Summation - product – transposition – square matrix)

1. Create **a new script** (Ctrl +N). Save with the name : **EX3_TP4.m**
2. Introduire les matrices dans le script.

3. Vérifier les matrices par l'exécution de programme  .
4. Calculer les dimensions de $A, A1, B, V1, V2\ et\ V3$.
5. Calculer la matrice transposée de A et B : **transpose** (A), B'.
6. Calculer la matrice carrée de B : **B^2**.
7. A partir de la matrice A, extraire la sous matrice $A2\ = (3{,}5\ ;\ 4{,}9)$
8. Calculer la somme des matrices A et A1 : $Som = A + A1$;
9. Calculer la somme des matrices B et A2 : $S = B + A2$;
10. Calculer la somme des matrices A et C3 : $Som1 = A + C3$;

```
>> Som1=A+C3
Error using  +
Matrix dimensions must agree.
```

Here's how you can perform the operations described in Exercise 4 using MATLAB:

Table 3. Exercise 4 (Summation - product – transposition – square matrix)

```
% Display matrices and vectors
disp("Matrix A:");
disp(A);
disp("Matrix A1:");
disp(A1);
disp("Matrix B:");
disp(B);
disp("Vector V1:");
disp(V1);
disp("Vector V2:");
disp(V2);
disp("Vector V3:");
disp(V3);

% Calculate dimensions
dim_A = size(A);
dim_A1 = size(A1);
dim_B = size(B);
dim_V1 = length(V1);
dim_V2 = length(V2);
dim_V3 = length(V3);
disp("Dimensions of A:");
disp(dim_A);
disp("Dimensions of A1:");
disp(dim_A1);
disp("Dimensions of B:");
disp(dim_B);
disp("Dimensions of V1:");
disp(dim_V1);
disp("Dimensions of V2:");
disp(dim_V2);
disp("Dimensions of V3:");
disp(dim_V3);

% Calculate transposed matrices
transpose_A = transpose(A);
transpose_B = B';
% Calculate square matrix
B_squared = B^2;
% Extract submatrix A2 from A
A2 = A(2:3, 1:2);
% Calculate matrix sum
Sum_A_A1 = A + A1;
Sum_B_A2 = B + A2;

% Calculate sum with incompatible
dimensions (A + C3)
C3 = [1 2 3; 4 5 6; 7 8 9];
Sum_A_C3 = A + C3;
```

Copy and paste the above code into a new MATLAB script file (e.g., "EX3_TP4.m") and run it to perform the calculations and operations as described in the exercise.

To calculate the sum of the two matrices, the dimensions of the matrices must be the same.

- Calculate the product of the matrices A and V1: $prd = A * V1$;

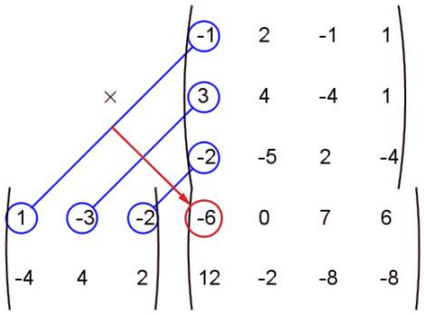

Figure 2. Matrix Product

For the product, the dimension of the column of the first matrix must be equal to the dimension of the row of the second matrix $(\boldsymbol{n}, \boldsymbol{m}) * (\boldsymbol{n}', \boldsymbol{m}') \gg \boldsymbol{m} = \boldsymbol{n}'$.

- Manually check the result.

```
>> prd=A*V1

prd =

    80
    77
    68
```

Figure 3. Matrix Product

Exercise 5 (specific matrices)

Using Matlab (Command Window space);

1. *Identity matrix: use the command **eye(n,m)**: 1) (n=3,m=1), 2) (n=3,m=2), 3) (n=3,m=3).*
2. *Null matrix: use the command **zeros(n,m)**: 1) (n=3,m=1), 2) (n=3,m=2), 3) (n=3,m=3).*
3. *Unit matrix: use the command **ones(n,m)**: 1) (n=3,m=1), 2) (n=3,m=2), 3) (n=3,m=3).*
4. *Random matrix: use the command **rand(n,m)**: 1) (n=3,m=1), 2) (n=3,m=2), 3) (n=3,m=3).*
5. *Magic Matrix: use the **magic(n)** command.*

Here's how you can create and work with different types of matrices using MATLAB:

% Matrice d'identité *identity_1 = eye(3, 1);* *identity_2 = eye(3, 2);* *identity_3 = eye(3, 3);* *disp("Identity Matrix (n=3, m=1):");* *disp(identity_1);*	*% Matrice unitaire* *ones_matrix_1 = ones(3, 1);* *ones_matrix_2 = ones(3, 2);* *ones_matrix_3 = ones(3, 3);* *disp("Ones Matrix (n=3, m=1):");* *disp(ones_matrix_1);*

```
disp("Identity Matrix (n=3, m=2):");
disp(identity_2);
disp("Identity Matrix (n=3, m=3):");
disp(identity_3);

% Matrice nulle
zero_matrix_1 = zeros(3, 1);
zero_matrix_2 = zeros(3, 2);
zero_matrix_3 = zeros(3, 3);
disp("Zero Matrix (n=3, m=1):");
disp(zero_matrix_1);
disp("Zero Matrix (n=3, m=2):");
disp(zero_matrix_2);
disp("Zero Matrix (n=3, m=3):");
disp(zero_matrix_3);
```

```
disp("Ones Matrix (n=3, m=2):");
disp(ones_matrix_2);
disp("Ones Matrix (n=3, m=3):");
disp(ones_matrix_3);

% Matrice aléatoire
random_matrix_1 = rand(3, 1);
random_matrix_2 = rand(3, 2);
random_matrix_3 = rand(3, 3);
disp("Random Matrix (n=3, m=1):");
disp(random_matrix_1);
disp("Random Matrix (n=3, m=2):");
disp(random_matrix_2);
disp("Random Matrix (n=3, m=3):");
disp(random_matrix_3);

% Matrice Magique
magic_matrix = magic(3);
disp("Magic Matrix (n=3):");
disp(magic_matrix);
```

Exercise 6 (introducing a matrix (2,2))

1. Create a new script (Ctrl+N). Save with the name: EX5_TP4.m
2. Create a function that takes 4 numbers as inputs and returns an M(2,2) matrix as output.
3. Use the command zeros (n,m) to initiate the matrix M(2,2).

Here's an example of how you can create a MATLAB script to define a function that takes four numbers as input and returns a 2x2 matrix:

```
% Define the function to create a 2x2 matrix
function M = createMatrix(a, b, c, d)
% Initialize a 2x2 matrix with zeros
M = zeros(2, 2);
% Assign the input values to the matrix elements
M(1, 1) = a;
M(1, 2) = b;
M(2, 1) = c;
M(2, 2) = d;
end
% Test the function with example values
a = 1;
b = 2;
```

14. c = 3;
15. d = 4;
16. result_matrix = createMatrix(a, b, c, d);
17. % Display the result matrix
18. disp("Resulting 2x2 Matrix:");
19. disp(result_matrix);

Then, run the script to define the function and test it with the provided example values. The function create Matrix takes four input values a, b, c, and d, and constructs a 2x2 matrix using the zeros function to initialize the matrix and assigns the input values to its elements. Finally, the resulting matrix is displayed using the ***disp*** function.

Exercice 7 (introduire une matrice (2,2) par l'utilisateur)

- Create **a new script** (Ctrl +N). Save with the name : **EX6_TP4.m**
- Réaliser une fonction qui prend 4 nombres en entrées par l'utilisateur et renvoie en sortie une matrice $M1(2,2)$. $M1 = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$

Utiliser la commande $m_{11} = \boldsymbol{input}$ (‘$entre\ le\ 1er\ nombre$ ‘) ;
Utiliser la commande $\boldsymbol{zeros}(\boldsymbol{n}, \boldsymbol{m})$ pour initier la matrice $M1(2,2)$.

Exercice 7 (introduire une matrice (2,2) par l'utilisateur)

1. Create a new script (Ctrl +N). Save with the name : EX6_TP4.m
2. Réaliser une fonction qui prend 4 nombres en entrées par l'utilisateur et renvoie en sortie une matrice $M1(2,2)$. $M1 = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$
   - Utiliser la commande m_11=input (‘entre le 1er nombre ‘) ;
   - Utiliser la commande $zeros(n, m)$ pour initier la matrice M1(2,2).

Here's how you can create a MATLAB script to define a function that takes four numbers as input from the user and returns a 2x2 matrix:

1. % Define the function to create a 2x2 matrix from user input
2. function M1 = createMatrixFromUserInput()
3. % Prompt the user to enter four numbers
4. m_11 = input('Enter the 1st number: ');
5. m_12 = input('Enter the 2nd number: ');
6. m_21 = input('Enter the 3rd number: ');
7. m_22 = input('Enter the 4th number: ');
8. % Initialize a 2x2 matrix with zeros
9. M1 = zeros(2, 2);
10. % Assign the user input values to the matrix elements
11. M1(1, 1) = m_11;
12. M1(1, 2) = m_12;
13. M1(2, 1) = m_21;

```
14. M1(2, 2) = m_22;
15. end
16. % Call the function to create the matrix from user input
17. result_matrix = createMatrixFromUserInput();
18. % Display the result matrix
19. disp("Resulting 2x2 Matrix:");
20. disp(result_matrix);
```

When you run the script, the function `createMatrixFromUserInput` prompts the user to enter four numbers. It then constructs a 2x2 matrix using the user's input values and the `zeros` function to initialize the matrix. Finally, the resulting matrix is displayed using the `disp` function.

Exercise 8 (introducing a matrix (3,3) by the user)

1. Create a new script (Ctrl+N). Save with the name: EX7_TP4.m
2. Create a function that takes 9 numbers as inputs by the user and returns a $M2(3,3)$ matrix as output.

- Use the command $a_{11} = input\ ('enter\ the\ 1st\ number')$;
- Use the command zeros(n,m) to initiate the matrix M2(3,3).

Exercise 9 (Summation Matrix(2,2) + Matrix(2,2) by choice)

1. We are looking to create a function that sums two matrices A + B.
2. Create a new script (Ctrl+N). Save with the name: EX8_TP4.m

Here's how you can create a MATLAB script to define a function that adds two 2x2 matrices and returns the result:

```
1. % Define the function to add two 2x2 matrices
2. function sum_matrix = addMatrices(matrixA, matrixB)
3. if size(matrixA) == [2, 2] && size(matrixB) == [2, 2]
4. % Perform matrix addition
5. sum_matrix = matrixA + matrixB;
6. else
7. disp('Both input matrices should be 2x2.');
8. sum_matrix = [];
9. end
10. end
11. % Example matrices A and B
12. A = [1, 2; 3, 4];
13. B = [5, 6; 7, 8];
14. % Call the function to add matrices A and B
15. result_matrix = addMatrices(A, B);
16. % Display the result matrix
```

```
17. disp("Resulting Sum Matrix:");
18. disp(result_matrix);
```

This script defines a function `addMatrices` that takes two input matrices and returns their sum if both matrices are 2x2. The example matrices A and B are provided, and the function is called to compute their sum. The result is displayed using the `disp` function.

Exercise 10 (Matrix(3,3) x Vector(3,1) multiplication by choice)

1. Create a new script (Ctrl+N). Save with the name: EX9_TP4.m
2. We are looking to create a function that produces the product between matrix(3,3) and a vector(3,1) entered by the user and displays the result as output.

- Use the command b11=input ('enter the 1st number');
- Use the command zeros(n,m) to initiate the matrix M2(3,3).

here's how you can create a MATLAB script for Exercise 10, where you'll implement a function that performs the multiplication of a 3x3 matrix and a 3x1 vector entered by the user:

```
1.  % Define the function to perform matrix-vector multiplication
2.  function result_vector = matrixVectorMultiplication(matrix, vector)
3.  if size(matrix) == [3, 3] && size(vector) == [3, 1]
4.  % Perform matrix-vector multiplication
5.  result_vector = matrix * vector;
6.  else
7.  disp('Matrix should be 3x3 and vector should be 3x1.');
8.  result_vector = [];
9.  end
10. end

11. % Prompt the user to enter matrix elements
12. matrix = input('Enter a 3x3 matrix [a11, a12, a13; a21, a22, a23; a31, a32, a33]: ');

13. % Prompt the user to enter vector elements
14. vector = input('Enter a 3x1 vector [b1; b2; b3]: ');

15. % Call the function to perform matrix-vector multiplication
16. result = matrixVectorMultiplication(matrix, vector);

17. % Display the result vector
18. disp('Resulting Vector:');
19. disp(result);
```

This script defines a function `matrixVectorMultiplication` that takes a 3x3 matrix and a 3x1 vector as input and returns their multiplication result. The user is prompted to enter the matrix and vector elements, and the function is called to compute the multiplication. The result is displayed using the `disp` function.

Part 03: Experimental part (MATLAB – SIMULINK)

Exercise 11 (solving a matrix linear system AX=Y)

1. Create a new script (Ctrl+N). Save with the name: EX10_TP4.m
2. Create a function that solves the matrix equation AX=Y and displays the result of the X as output, with the matrix A(3,3), the vectors Y(3,1), X(3,1).
3. Measure the execution time of the program in seconds; use command tick; and knock;

- Use the command a11=input ('enter the 1st number');
- Use the command zeros(n,m) to initiate the matrix A(3,3).
- Use the command zeros(n,m) to initiate the vector X(3,1).
- Use the zeros(n,m) command to initiate the Y(3,1) vector.

Exercise 12 (determinant of a matrix)

1. Create a new script (Ctrl+N). Save with the name: EX11_TP4.m
2. Write a function that returns the determinant of an N(2,2) matrix given by the user.

- Use the command var11=input ('enter the 1st number');
- Use the zeros(n,m) command to initiate the N(2,2) matrix.

Exercise 13 (inverse of a matrix (2,2))

1. Create a new script (Ctrl+N). Save with the name: EX12_TP4.m
2. Write a function that inverts an L(2,2) matrix given by the user.

- Use the command a11=input ('enter the 1st number');
- Use the zeros(n,m) command to initiate the L(2,2) matrix.

Exercise 14 (inverse of a matrix (3,3))

1. Create a new script (Ctrl+N). Save with the name: EX12_TP4.m
2. Realize a function that inverts a matrix G(3,3) given by the user.

- Use the command G11=input ('enter the 1st number');
- Use the command zeros(n,m) to initiate the matrix G(3,3).

2.11. Conclusion

This chapter provided a comprehensive understanding of the various data types and variables used in MATLAB. You have gained skills in manipulating complex numbers, Boolean variables, vectors, matrices, and polynomials, all of which are essential for performing mathematical calculations and simulations. By mastering these concepts, you are now better equipped to tackle more complex programming tasks in MATLAB, thereby laying the foundation for efficient and effective problem-solving in engineering applications.