DOCUMENT FOR COURSE

Informatique 3

(Cours + Practical Work)

second-year civil engineering

HAMIDA SOURAYA
Associate Professor
2024-2025

Table of Contents

Preface

This course was created especially to give second-year civil engineering students a hands-on understanding of MATLAB, a crucial tool for resolving challenging engineering science problems. Because of its extensive application in data analysis, modelling, and numerical computation, MATLAB is a useful tool for managing civil engineering projects. This course seeks to improve students' technical proficiency while equipping them to apply MATLAB in practical and diverse applications through concrete examples. The objective is to enable each student to use MATLAB independently for professional projects.

Chapter 1: Presentation of a Scientific Programming Environment

1 Introduction

2 Some Definitions

3 MATLAB

4 MATLAB as a calculator

5 Help

6 Interaction Commands

1 Introduction

In this chapter, you will delve into understanding the essentials of programming and how MATLAB, a powerful tool for numerical computing, plays a pivotal role in various fields. This chapter aims to provide a solid foundation in programming, offering clear definitions and explanations of key terms and concepts. This chapter will guide you through the basics and prepare you for more advanced topics.

2 Some Definitions

2.1 Programming

A computer can only perform very basic operations:

- Adding, subtracting, multiplying, dividing numerical values
- Moving data from one part of its memory to another
- Jumping from one line of code to another if a condition is true

Programming involves combining these instructions that act on data to accomplish a specific task using the computer (e.g., testing if a number is prime, detecting elements in an image, etc.)

2.2 Programming Language

A programming language is a way to write instructions that will then be "translated" into basic operations for the computer.

Instructions generally have a specific task; together they form software.

2.3 Software

Software is a series of instructions written in a programming language that allows one or more tasks to be accomplished (text editor, video games, video player, internet

browser, etc.). Most of the time, it has a graphical interface to facilitate interactions with the user.

2.4 Framework

A framework is a collection of subprograms written in a specific language, enabling complex tasks to be performed more simply.

A framework allows you to reuse the work done by others for your program, saving time by avoiding writing what others have already written.

3 MATLAB

3.1 Definition

MATLAB (Matrix Laboratory) is a numerical computation software developed by MathWorks, initially designed by Cleve Moler in the late 70s to offer a high-level programming language without requiring the learning of Fortran or C.

It has an interactive interface for running commands and an integrated development environment (IDE) for creating applications.

MATLAB is widely used across various disciplines for physical system modelling, mathematical model simulation, design and validation (tests in simulation and experimentation) of applications. Additionally, it can be extended with toolboxes, and specialized libraries for fields such as automation, signal processing, statistical analysis, and optimization.

3.2 Particularities of MATLAB

MATLAB offers an interactive environment that allows working in command mode or programming mode, with the constant possibility of generating graphical visualizations. Recognized as a powerful programming language, MATLAB

distinguishes itself by the following characteristics compared to languages like C or Fortran:

- Easy and intuitive programming
- Transparent handling of integers, real numbers, and complex numbers
- Wide range of precision and number extent
- A very comprehensive mathematical library
- The graphical tool includes graphical interface functions and utilities
- The possibility of interfacing with other traditional programming languages (C or Fortran)

Number Management without Prior Declaration

In MATLAB, no declaration is required for numbers. Indeed, there is no distinction between integers, real numbers, complex numbers, and single or double precision. This feature makes the programming mode very easy and fast.

Mathematical Library

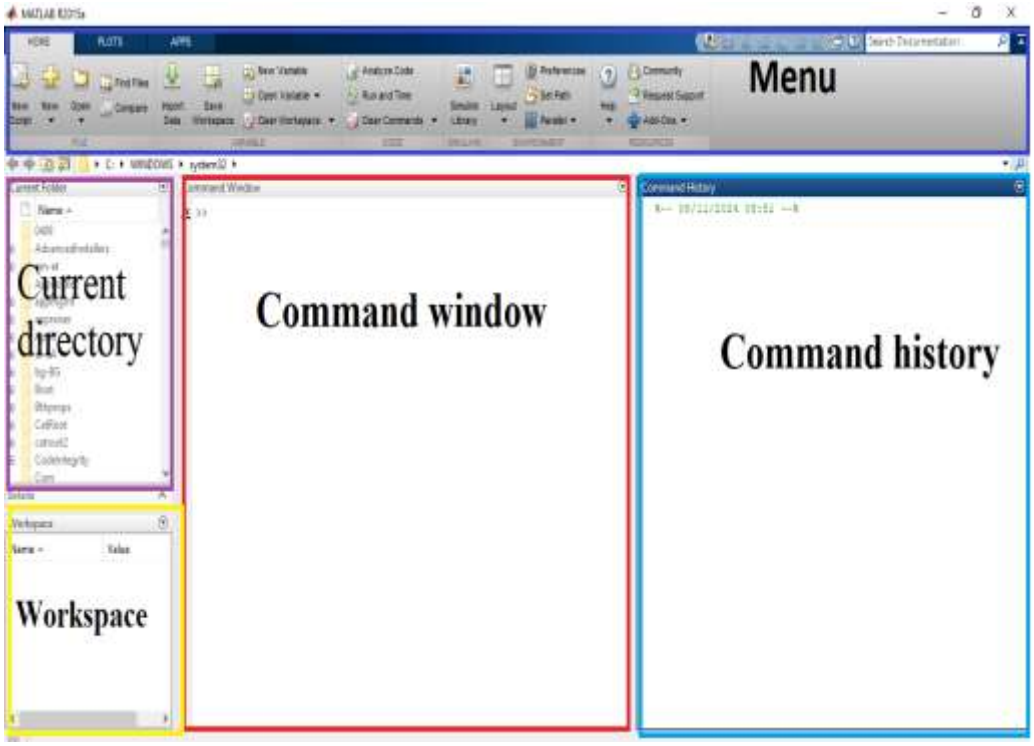
The mathematical function library in MATLAB provides very simple mathematical analyses. Indeed, the user can execute any mathematical function in the library in command mode without resorting to programming.

Graphical Tools

For the graphical interface, scientific and even artistic representations of objects can be created on the screen using mathematical expressions. The graphs on MATLAB are simple and attract the attention of users, given the important possibilities offered by this software.

3.3 **Main interface**

Depending on the version used, the interface may change slightly but the central points will remain the same. When launching Matlab, the following interface appears:



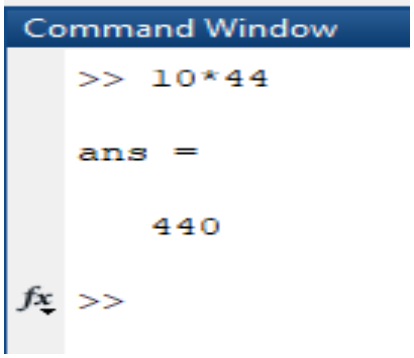
The software offers a real working environment composed of multiple windows. We can distinguish four blocks:

- Command window: at the command prompt “>>”, the user can enter the instructions to execute. This is the main window of the interface.
- Current directory: allows you to navigate and view the contents of the user’s current directory. The user’s programs must be located in this directory to be visible and therefore executable.
- Workspace: allows you to view the defined variables, their type, the size occupied in memory, etc.

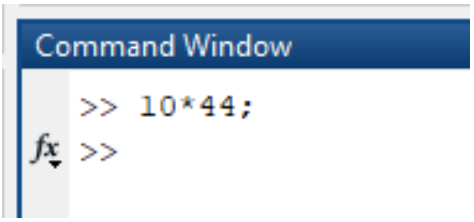
- Command history: history of commands that the user has executed. It is possible to drag these commands to the command window.

Note that the command window is the central window of the interface, and from it, the user can run commands that Matlab interprets.

The principle is simple and intuitive, the trick is to know the appropriate functions and respect their syntax. First elementary example: at the command prompt, type "10*44", then enter:



When the instruction is validated, the interface displays the result of the instruction. To simplify the display, a semicolon ";" at the end of the command prevents the result from being returned to the window (obviously the instruction is still executed). For example:



The calculation was performed but the result is not displayed.

4 MATLAB as a calculator

As an example of a simple interactive calculation, just type the expression you want to evaluate. Let's start at the very beginning. For example, let's suppose you want to calculate the expression, 1 + 2 × 3. You type it at the prompt command (>>) as follows,

```
>> 1+2*3

ans = 7
```

You will have noticed that if you do not specify an output variable, MATLAB uses a default variable ***ans***, short for the ***answer***, to store the results of the current calculation. Note that the variable ans is created (or overwritten, if it is already existed). To avoid this, you may assign a value to a variable or output argument name. For example,

```
>> x = 1+2*3

x= 7
```

will result in x being given the value 1 + 2 × 3 = 7. This variable name can always be used to refer to the results of the previous computations. Therefore, computing 4x will result in

```
>> 4*x

ans = 28.0000
```

5 Help

When using a high-level programming language (such as MATLAB), where the syntax might be complicated and the number of functions is substantial, it is crucial to use help.

MATLAB help can be shown with the ***help*** command. However, the FunctionName function's syntax and explanation can be shown with the ***help FunctionName*** command.

6 Interaction Commands

Command	**Objective**
Who	Displays the names of variables in the Workspace.
Whos	Displays information about variables in the Workspace.
Clear x y	Removes the x and y variables from the Workspace.
Clear, clear all	Removes all variables in the Workspace.
Clc	Clears the command screen in the Command Window.
Exit, quit	Exit MATLAB.
format	Sets the output format for numeric values

Chapter 2: Script Files and Data Types and Variable

1 introduction

2 Script Files

3 Data Types in MATLAB

4 Variables

5 Comparison operators and logical operators

6 Arithmetic and operations on scalars

1 Introduction

This chapter will provide you with the knowledge and tools necessary to create robust scripts, manage diverse data types, and efficiently use variables in your MATLAB projects.

2 Script Files

A script is the simplest .m file. It is simply a list of commands put together and saved in a file. The M-files can be scripts that simply execute a series of instructions or can be functions. The Scripts make it easy to automate repetitive tasks.

2.1 Advantages of scripts

- Reuse of code.
- Organization of commands to facilitate modifications and maintenance.
- Ability to work with variables defined in the MATLAB workspace.

2.2 Creating a script

- Use the MATLAB editor to create a script. Save the file with the `.m` extension. Example: `my_script.m`.
- To run the script, type its name without the extension in the MATLAB console, for example: `my_script`.
- From Matlab, an m-file is created or opened, either from the File menu (New > M-File) or the prompt by typing: >> edit myfile.m
- A m-file is recognized, and therefore executable, if it is in the current directory or if the containing directory is specified in the PATH.

2.3 Comments

Use `%` to add a comment on a line.

Example:

```
% This is a comment

x = 15; % Initialization of variable x
```

2.4 Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

- Variables already existing in the workspace may be overwritten.
- The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side-effects, it is better to code any complicated applications using rather function M-file.

3 Data Types in MATLAB

3.1 The 5 MATLAB Data Types

MATLAB supports several types of variables, including:

- Integers
- Real: Numerical values without an imaginary part.
- Complex: Numbers with both real and imaginary parts.
- String: Represented by an array of characters.
- Logical: For Boolean values (0 for false, 1 for true).

The type declaration is not necessary; it is automatically assigned according to the assigned value.

For example, the instructions x = 2; z = 2+i; rep = 'yes'; define a variable x of type real, a variable z of type complex and a variable rep of type string.

```
Command Window
>> x = 2; z = 2+i; rep = 'yes'; e = int8(2);
>> whos
  Name       Size              Bytes  Class     Attributes

  e          1x1                   1  int8
  rep        1x3                   6  char
  x          1x1                   8  double
  z          1x1                  16  double    complex

fx >>
```

3.1.1 The String Type

A string is an array of characters and can be manipulated like a normal array. The following example shows different manipulations of a string.

```
Command Window
>> ch1 = 'good';
>> ch2 = 'day';
>> ch = [ch1, ch2]; % ch becomes 'goodday'
fx >>
```

To manipulate individual characters, we use indices:

```
Command Window
>> ch(1)        % 'g'

ans =

g

>> ch(1:4)      % 'good'

ans =

good
```

If a string contains apostrophes, they must be doubled to be included correctly:

```
rep = 'aujourd"hui';    % aujourd'hui
```

3.1.2 Logical type

Logical variables are used in tests and return 0 for false and 1 for true.

Example:

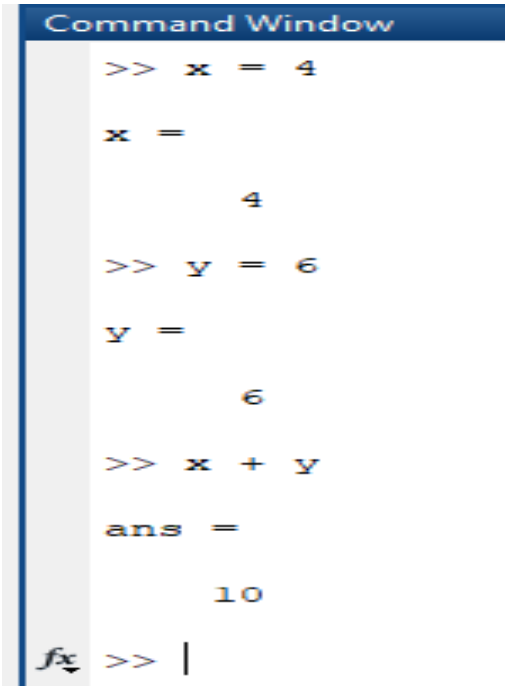
```
Command Window
>> x = 123; y = exp(log(x));
>> test = (x == y)

test =

     0
```

test will be 0 because the two are not exactly equal

4 Variables

4.1 Elementary aspects

Matlab automatically manages integers, real numbers, complex numbers, character strings, etc. Thus, the declaration of variables is implicit, and the necessary memory is automatically allocated when defining them. The assignment symbol is the "=" sign.



Regarding variable names, the interpreter distinguishes between lowercase and uppercase letters. When a variable is defined, it appears, along with some information, in the Workspace window.

4.2 **Predefined constants**

Some symbols have predefined values associated with them. Here are some of them:

Symbol	**Meaning**	**Value**
Pi	π	3.141592...
i or j	Complex number	$\sqrt{-1}$
realmax	Largest encodable floating point number	1.7977e+308
realmin	Smallest encodable floating point number	2.2251e-308

5 Comparison operators and logical operators

The comparison operators are:

- = = : equal to (X= = Y)
- >: Strictly greater than (X > Y)
- < : Strictly less than (X< Y)
- >= : greater than or equal to (X>= Y)
- <= : less than or equal to (X<= Y)
- ~= : different from (X~=Y)

The logical operators are:

- & : and (X &Y)
- | : or (X | Y)
- ~: Not (not) X (~X)

6 Arithmetic and operations on scalars

We will focus on basic mathematical operations with 1x1 matrices, that is, numbers. Let's start with the 4 operations that you know from primary school: +,- ,*,/

```
>> x = 1+1
x=2
```

We can also use trigonometry, power, logarithmic functions, etc.

```
>> x = 2; y = pi;
cos(y)
exp(x)
sqrt(x)
ans = -1
ans = 7.3891
ans = 1.4142
```

Here is a list (not exhaustive) of the functions incorporated in Matlab:

- exp(x): exponential of x
- log(x): natural logarithm of x
- log10(x): logarithm in base 10 of x
- xˆn: x to the power n
- sqrt(x): square root of x
- abs(x): absolute value of x
- sign(x): 1 if x>0 and 0 if x < 0
- sin(x): sine of x
- cos(x): cosine of x
- tan(x): tangent of x

We can also use the rounding functions:

- round(x): integer closest to x
- floor(x): rounding down from
- x ceil(x): rounding up from x

As well as arithmetic functions:

- rem(m,n): remainder of the integer division of m by n
- lcm(m,n): least common multiple of m and n
- gcd(m,n): greatest common divisor of m and n
- factor(n): prime factorization of n

Finally, when working with complex numbers, we can use:

- conj(z): conjugate of z
- abs(z): modulus of z
- angle(z): argument of z
- real(z): real part of z
- imag(z): imaginary part of z

Chapter 3: Reading, displaying and saving data

1 Introduction

This chapter provides an essential guide to understanding how to effectively read, visualize, and save data in MATLAB. These tools will enhance your scripts by improving interactivity and data presentation.

2 Lecture

The **input** command is used to ask the user of a program to provide data.

Syntax:

```
var = input(' a sentence ')
```

The sentence: a sentence is displayed and MATLAB waits for the user to enter data on the keyboard. This data can be a numeric value or a MATLAB instruction. A carriage return causes the input to end.

A numeric value is directly assigned to the variable var while a MATLAB instruction is evaluated and the result is assigned to the variable var.

It is possible to cause line breaks to air the presentation by using the \n symbol in the following way:

```
var = input('\n a sentence : \n ')
```

In this form, it is impossible to have a string-type data since MATLAB tries to interpret this string as an instruction. If we want to enter a string-type response we use the syntax:

```
var = input(' a sentence ','s')
```

Example:

```
rep = input('Display result? y/n [y] ','s');
if isempty(rep), rep = 'y'; end
if rep == 'y' | rep == 'n'
disp(['Result is ', num2str(rep)])
end
```

3 Real number display formats

MATLAB has several real-number display formats. By default, the format is the short 5-digit format. The other main formats are:

long format	15-digit long format.
short format **e**	5-digit short format with floating point notation.
long format **e**	15-digit long format with floating point notation.

MATLAB also has the formats **format short g** and **format long g** which use the "best" of the two **fixed-point** or **floating-point formats**.

All possible display formats can be obtained by typing **help format**. A display format is imposed by typing the corresponding format instruction in the control window, for example, **format long**. To return to the default format, use the **format** or **format short** command.

Example :

```
>> pi
ans =
   3.1416
>> format long
>> pi
ans =
  3.14159265358979
>>  format short e
>> pi^3
ans =
  3.1006e+01
>> format short g
>> pi^3
```

```
ans =
      31.006
>> format short
```

4 Simple display, the disp command

The disp command allows you to display a table of numeric or character values. The other way to display a table is to type its name. The disp command simply displays the table without writing the name of the variable, which can improve certain presentations.

Example :

```
>> A = magic(4);
>>  disp(A)
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
>> A
A =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
>>
```

The disp command is frequently used with an array which is a string of characters to display a message.

Example :

```
disp(['The determinant of matrix A is ', num2str(det(A))]).
```

We note that the use of the disp command is then a little particular. Indeed, an array must be of a given type, the elements of the same array cannot therefore be character strings and numeric values.

We therefore use the **num2str** command (<< number to string >>) to convert a numeric value into a character string.

By default, the num2str command displays 4 decimal places, but it is possible to specify the desired number of decimal places in the second parameter. Similarly, it is possible to specify a particular display format for the numeric value.

5 sprintf command

The sprintf command allows variables to be printed according to a given template. An edit template is presented in the form of the percent symbol (%) followed by indications allowing the contents of the field to be printed to be composed, in particular its length in the number of characters.

Syntax

---

sprintf(format, variables)

---

where

- **variables**: is the name of the variables to be printed according to the editing template specified in format;
- **format**: is the editing format. This is a string containing the editing templates of the variables to be printed.

5.1 Character Editing Model

A character editing template is of the form **%Ls** where:

- **%** is the start of the format symbol
- **s** the symbol specifying that the data is of type character string.
- **L** is an integer giving the total length of the field (in number of characters). By default, the field is justified on the right (if the length of

the character string is smaller than the length L of the field, spaces are inserted after the character string).

- The symbol - (minus) just after the % symbol allows left-justified.
- In the absence of the integer **L** the total length of the field is equal to the number of characters in the string.

```
Command Window
>> sprintf('%s', 'it will be nice in Biskra')

ans =

it will be nice in Biskra

>> weather = 'it will be nice in Biskra'; sprintf('%s', weather)

ans =

it will be nice in Biskra

>> sprintf('%30s', weather)

ans =

     it will be nice in Biskra

>> sprintf('%-30s', weather)

ans =

it will be nice in Biskra

>> sprintf('weather : %s', weather)

ans =

weather : it will be nice in Biskra
```

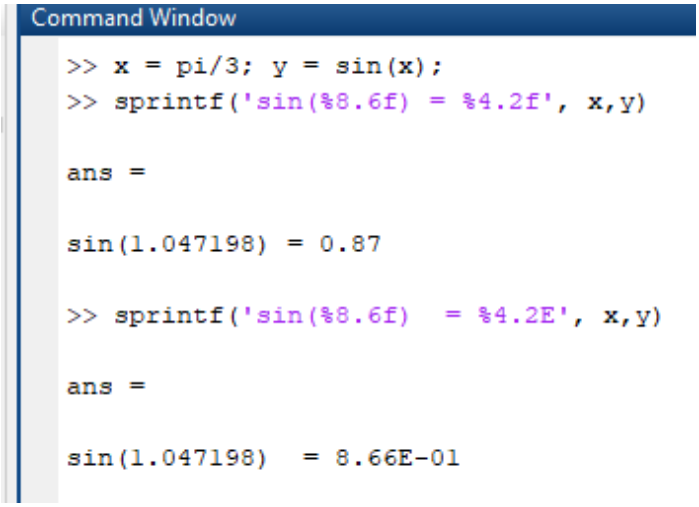
5.2 Real number editing template

A real number editing template is of the form **%+- L.D t**, where

- **%** is the start of the format symbol,

- **L** is an integer giving the total length of the field (in number of characters, including semicolon),
- **D** is the number of decimal places to display
- **t** specifies the type of notation used. By default, the field is right-justified (if the length of the variable is smaller than the length of the field L, spaces are inserted on the left).
- The - (minus) symbol is used to justify on the left.
- The + (plus) symbol causes a **+ sign** to be systematically displayed in front of positive real numbers.
- The main possible values for **t** are:
    - **d** for integers
    - **e** for floating point notation where the exponent part is delimited by a lowercase e (eg: 3.1415e+00)
    - **E** same notation but E replaces e (eg: 3.1415E+00)
    - **f** for fixed point notation (eg: 3.1415)
    - **g** the most compact notation between floating point and fixed point notation is used

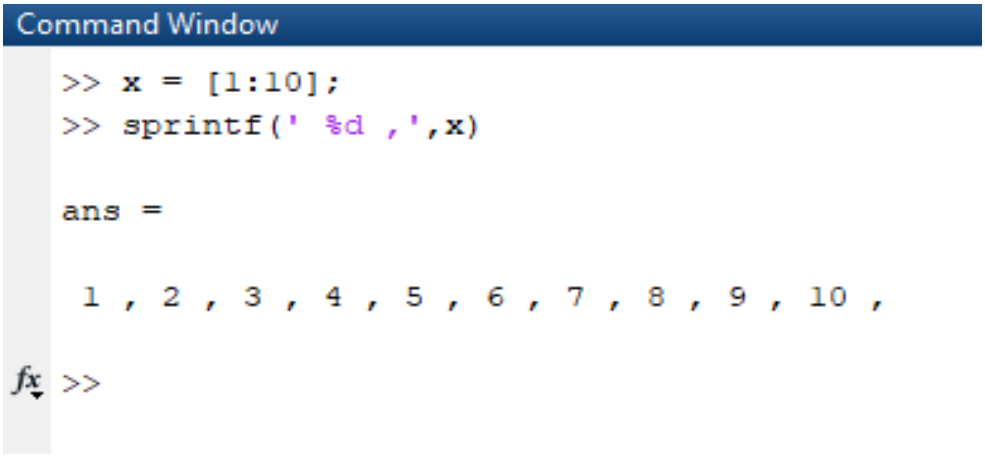
Example:



5.3 **Special uses**

The sprintf command is << vector >>: if the variable is not scalar the print format is reused for all the elements of the table, column by column.

Example:



It is possible to use the following symbols in strings

• \n: causes a new line

• \t: inserts a horizontal tab

• \b: shifts the printing of the next field by one character to the left

• \r : horizontal break

Example:

```
Command Window
>> z =[]; x = [1:10];
>> for i=1:length(x)
 z = [z ,x(i), log(x(i))];
 end;
>> s = sprintf('%4.1f | %8.6E \n ', z )

s =

 1.0 | 0.000000E+00
  2.0 | 6.931472E-01
  3.0 | 1.098612E+00
  4.0 | 1.386294E+00
  5.0 | 1.609438E+00
  6.0 | 1.791759E+00
  7.0 | 1.945910E+00
  8.0 | 2.079442E+00
  9.0 | 2.197225E+00
 10.0 | 2.302585E+00
```

If we need to display the % character we will double it %% so that it is not interpreted as the beginning of a format.

The **fprintf** command is the analogue of sprintf to print variables according to a given pattern in a file.

6 File: Reading, displaying and saving data

6.1 fopen function

The fopen function in MATLAB is used to open a file and get a file identifier that allows reading from or writing to that file.

Syntax

fid = fopen(‘file_name', 'mode');

where

- file_name: The name of the file you want to open. This can include the full path if the file is not in the current working directory.
- mode: A string argument that specifies the access mode for the file. Common modes include:
    - 'r': Open the file as read-only.
    - 'w': Open the file for writing (and clear the contents if it already exists).
    - 'a': Open the file in append mode (and append to the end of the file if it already exists).
    - 'r+': Open the file as read/write.
    - 'w+': Open the file as read/write (and clear the contents if it already exists).
    - 'a+': Open the file as read/write append.

6.2 fclose function

The fclose function in MATLAB is used to close an open file, thereby releasing the resources associated with the file. It is essential to always close files after using them to avoid memory leaks and ensure that all data is properly saved.

Syntax

```
status = fclose(fid);
```

where:

- fid : The file identifier, obtained when opening the file with fopen.
- status : A success or failure indicator. 0 indicates that the file was closed successfully, and -1 indicates an error.

Note :

- fclose('all') : to close all open files

6.3 fread function

The fread function is used to read binary data from an open file. This function is particularly useful for reading non-text files, such as images or binary files generated by other programs.

Syntax:

A = fread(fid, size, precision, skip, machinefmt)

Where:

- fid : The file identifier, obtained when opening the file with fopen.
- size : Specifies the number of elements to read and their format. For example, [m, n] reads data in an m by n matrix.
- precision : Specifies the format of the data to read, such as 'int32', 'float', 'char', etc.
- skip : (Optional) Number of bytes to skip after each element read.
- machinefmt : (Optional) Machine architecture format, such as 'ieee-le' or 'ieee-be' (little-endian or big-endian).

6.4 fwrite function

The fwrite function is used to write binary data to a file. This function is especially useful when you want to save data in an efficient and compact format.

Syntax:

---

count = fwrite(fid, A, precision, skip, machinefmt)

Where:

- fid : The file identifier, obtained when opening the file with fopen.
- A : The data to write to the file.

- precision : Specifies the format of the data to write, such as 'int32', 'float', 'char', etc.
- skip : (Optional) Number of bytes to skip after each element written.
- machinefmt : (Optional) Machine architecture format, such as 'ieee-le' or 'ieee-be' (little-endian or big-endian).
- count : Returns the number of elements of A that have been written.

6.5 fprintf function

The fprintf function is used to write formatted data to a file or to display formatted data to the console. This function is very powerful for creating well-structured textual output.

fprintf(fid, formatSpec, A, ...)

- fid : The file identifier, obtained when opening the file with fopen. Use fid as 1 to output to the console.
- formatSpec : A string that specifies the output format.
- A : The data to write or display.

7 fscanf function

The fscanf function is used to read formatted data from an open text file. It is particularly useful for extracting information structured according to a specific format.

Syntax:

A = fscanf(fid, formatSpec, sizeA);

Where:

- fid : The file identifier, obtained when opening the file with fopen.

- formatSpec : A string that specifies the format of the data to be read, such as %d, %f, %s, etc.
- sizeA : Specifies the size of the array A that receives the read data (optional).

Chapter 4: Vectors and matrices

1 Introduction

Vectors and matrices are the foundations of computation in MATLAB. This chapter explores their creation, manipulation, and use in various contexts. We will discover how MATLAB facilitates operations on these essential data structures, making complex computations more accessible and efficient.

2 Vectors

2.1 Defining a Vector

Vectors are created by listing elements in square brackets ([ ]):

- Row vector: elements separated by spaces or commas.
- Column vector: elements separated by semicolons or newlines.

Example:

```
>> x1 = [1 2 3]; % Row vector

>> x3 = [8; 9; 10]; % Column vector
```

2.2 Manipulating the elements of a vector

- An element is accessed by specifying its index:
  - X = [1 2 3 4 5];
  - X(3) % Third element
- We can also extract several elements at once:
  - X(2:4) % Elements 2 to 4

2.2.1 Arithmetic sequence

We can generate vectors with an arithmetic progression using the syntax:

- x = a:h:b; % From a to b with a step of h

The linspace function creates a vector with linearly spaced elements:

x = linspace(1.1, 1.9, 9); % From 1.1 to 1.9 with 9 elements

2.3 Special vectors

Some commands allow to generate vectors with specific values:

- ones(1,n): row vector of n elements equal to 1.
- zeros(1,n): row vector of n elements equal to 0.
- rand(1,n): row vector of n random elements between 0 and 1.

3 Matrices

3.1 Define a matrix

A matrix is created by giving the elements in square brackets, with the lines separated by semicolons.

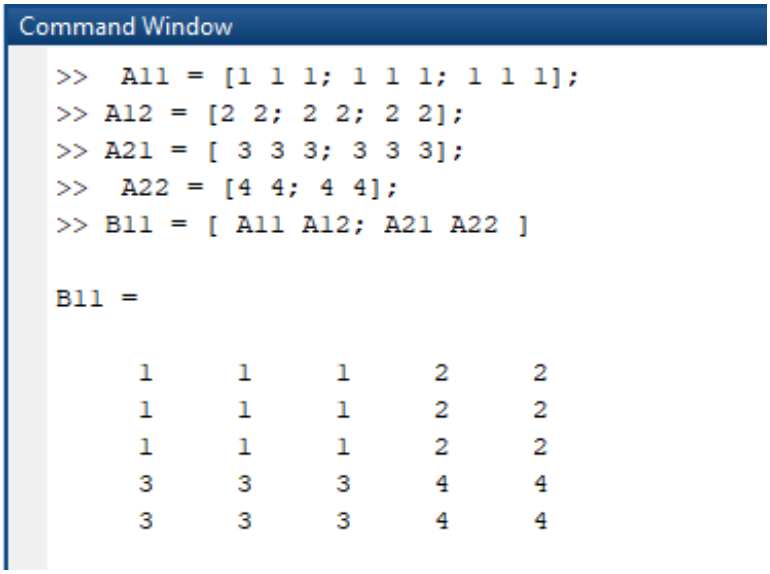
- Example: A = [1 3; 4 2]; % 2x2 matrix

We can also reference individual elements by their row and column indices:

- A(2,1) % First element of the second row (value 4)

We can construct a **block** matrix very simply. If A, B, C, D designate 4 matrices (with compatible dimensions), we define the block matrix, $B11 = \left(\begin{array}{c|c} A11 & A12 \\ \hline A21 & A22 \end{array}\right)$ by the instruction B11 = [A11 A12 ; A21 A22].

Example:



3.2 Special matrices

Some commands to generate special matrices:

- eye(n): identity matrix of dimension n.
- ones(m,n): m×n matrix with elements equal to 1.
- zeros(m,n): m×n matrix with elements equal to 0.
- rand(m,n): m×n matrix with random elements.
- magic(n): magic matrix of dimension n.

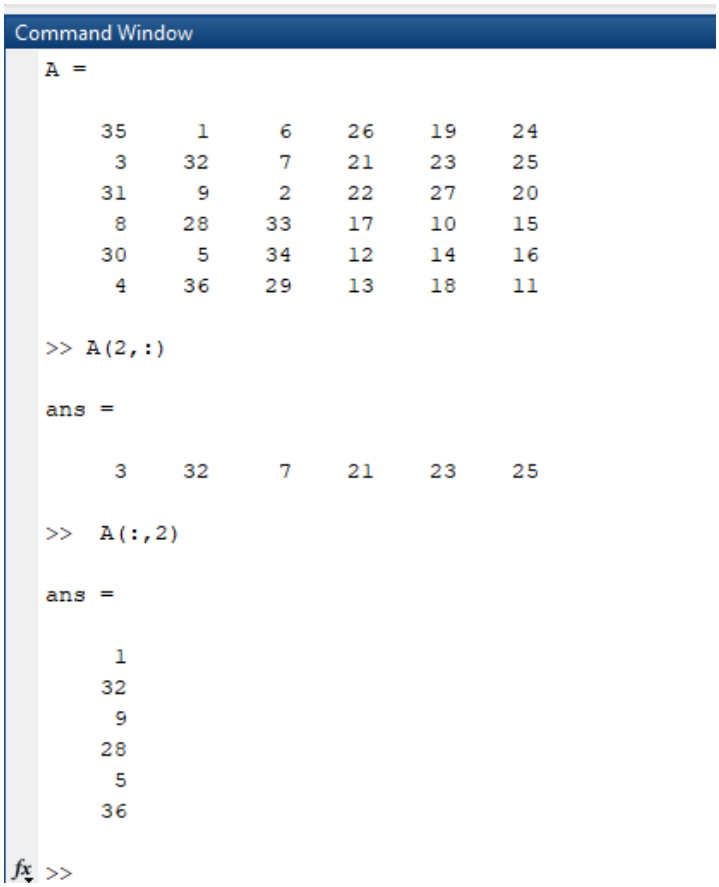
```
Command Window
>> eye(3)

ans =

     1     0     0
     0     1     0
     0     0     1

>> ones(3,2)

ans =

     1     1
     1     1
     1     1

>> zeros(2)

ans =

     0     0
     0     0
```

```
Command Window

>> rand(2,3)

ans =

    0.2785    0.9575    0.1576
    0.5469    0.9649    0.9706

>>  magic(5)

ans =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>>
```

3.3 Manipulating the elements of a matrix

Simple row or column extraction

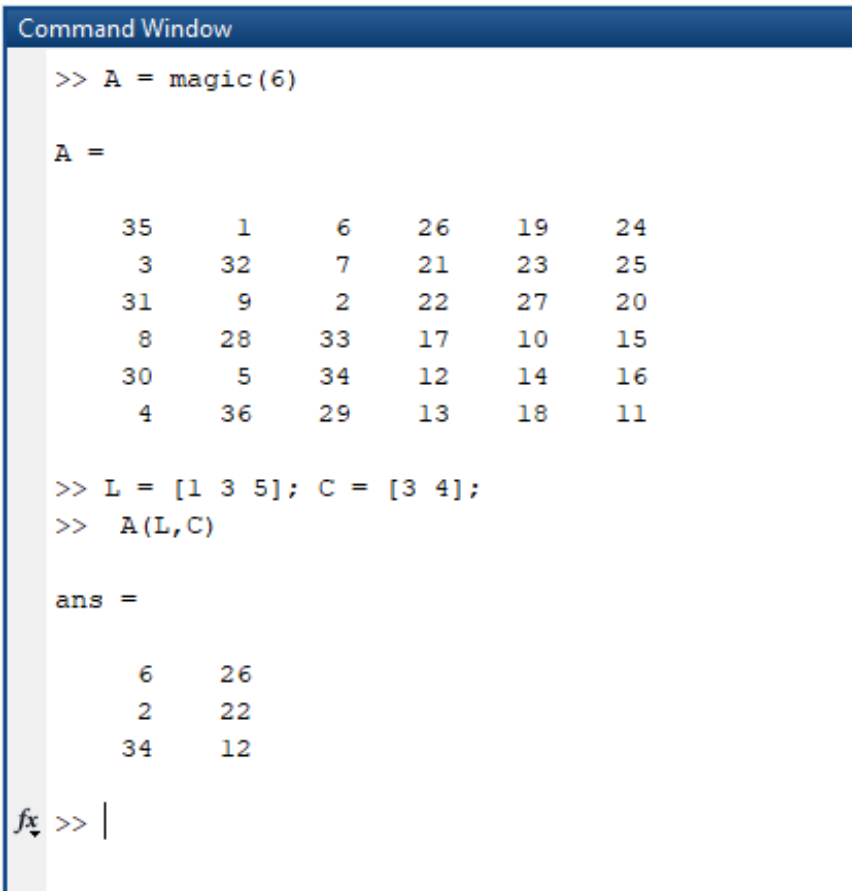
The colon symbol (:) is used to easily extract rows or columns from a matrix. The j^{th} column vector of matrix A is denoted by A(:,j). It's simple, just translate the colon symbol (:) as << all >>. The i^{th} row of matrix A is denoted by A(i,:).



Simultaneous extraction of multiple rows or columns

We can also extract several rows or columns simultaneously. If J is a vector of integers, A(:,J) is the matrix from A whose columns are the columns of the matrix A of indices contained in the vector J. Similarly, A(J,:) is the matrix from A whose rows are the rows of the matrix A of indices contained in the vector J. More generally, it is

possible to extract only part of the elements of the rows and columns of a matrix. If L and C are two vectors of indices, A(L,C) denotes the matrix from the matrix A whose elements are the A(i,j) such that i is in L and j is in C.



4 Matrix vs. vector Operators

Symbol	Operation	Symbol	Operation
+	Matrix addition	+	vector addition
-	Matrix subtraction	-	vector subtraction
*	Matrix multiplication	.*	vector multiplication
/	Matrix division	./	vector division
\	Left matrix division	.\	Left vector division
^	Matrix power	.^	vector power

5 Useful Functions

- length(A): Returns either the number of elements of A if A is a vector or the largest value of m or n if A is an m × n matrix
- size(A): Returns a row vector [m n] containing the sizes of the m × n matrix A.
- max(A): For vectors, returns the largest element in A. For matrices, returns a row vector containing the maximum element from each column. If any of the elements are complex, max(A) returns the elements that have the largest magnitudes.
- [v,k] = max(A): Similar to max(A) but stores the maximum values in the row vector v and their indices in the row vector k.
- min(A) and [v,k] = min(A) :Like max but returns minimum values.
- sort(A): Sorts each column of the array A in ascending order and returns an array the same size as A.
- sort(A,DIM,MODE): Sort with two optional parameters: DIM selects a dimension along which to sort. MODE is sort direction ('ascend' or 'descend').
- sum(A): Sums the elements in each column of the array A and returns a row vector containing the sums.
- sum(A,DIM): Sums along the dimension DIM.
- diag(A): is used to create a diagonal matrix or to extract the diagonal elements from a matrix or vector.
- tril: Returns the lower triangular part
- triu: Returns the upper triangular part
- det :Calculates the determinant of a matrix
- inv: Calculates the inverse of a matrix

Chapter 5: Control instructions

1 Introduction

MATLAB, as a high-level programming language, offers various control structures to manage the execution of commands. These structures allow the creation of dynamic and interactive programs. This chapter presents the main control structures in MATLAB.

2 Conditional Control Structures

2.1 Simple Conditional Control Structure (IF)

A block of statements can only be executed if the CONDITION IS TRUE (= the evaluation of a logical expression evaluates to TRUE), otherwise no statement is executed.

Syntax:

If *logical expression*

    sequence of statements

end

---

where

- *logical expression* is an expression whose result can be true or false;
- *sequence of statements* is the processing to be performed if the logical expression is true.

Interpretation:

The *sequence of statements* is executed only if the result of the evaluation of the logical expression is true (i.e., equals 1). Otherwise, the instruction following the end keyword is executed.

In the case where the logical expression is true, after executing the sequence of instructions, the program resumes the instruction following the end keyword.

2.2 Alternative conditional control structures

This structure allows the execution of a block of actions if a logical expression evaluates to TRUE and another block of actions otherwise.

Syntax:

---

if *logical expression*

statement sequence 1

else

statement sequence 2

end

---

where

- *logical expression* is an expression whose result can be true or false;
- *statement sequence 1* is the statement sequence to be executed if the logical expression is true
- *statement sequence 2* is the statement sequence to be executed if the logical expression is false.

Interpretation:

If the logical expression is true, statement sequence 1 is executed, otherwise, statement sequence 2 is executed. The program then resumes at the first instruction following the **end** keyword.

3 Repetitive control structures (loops)

3.1 FOR loop

The FOR loop is used to iterate over a block of statements a fixed number of times. It is often used when you know in advance the number of iterations you want to perform.

Syntax:

---

for index=lower bound : upper bound

instruction sequences

end

---

where

- *Index:* is a variable called the loop index;
- *lower bound and upper bound:* are two real constants (called loop parameters);
- *instruction sequence* is the processing to be performed for index values varying between the lower bound and upper bound with an increment of 1. This is called the body of the loop.

Interpretation*:*

- If the *lower bound is less than or equal to the upper bound*, the instruction sequence processing is executed (*upper bound - lower bound*) times, for values of the index variable equal to *lower bound*, *lower bound* +1, ..., *upper bound*.
- *If the lower bound is strictly greater than the upper bound*, we move on to the instruction immediately following the end of loop instruction (end).

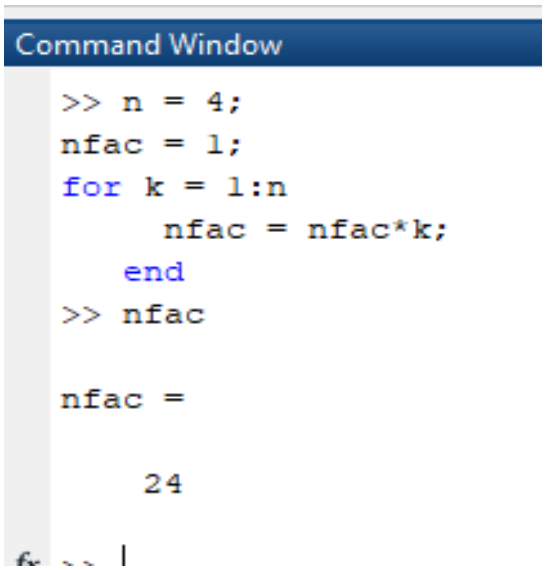
Note:

- The loop index does not necessarily take integer values.
- The loop index doesn't need to appear in the body of the loop; however, it is forbidden to modify its value if it appears.
- An increment (step) other than 1 (default value) can be used.
    - The syntax is then *lower bound*: step: *upper bound*.
        - The step can be negative.

Example 1

```
Command Window
>> for r=1.1:-0.1:0.8
     disp(['r = ', num2str(r)]);
   end
r = 1.1
r = 1
r = 0.9
r = 0.8
```

Example 2



3.2 WHILE loop

The repetitive control structure known as a WHILE loop allows a block of statements to be executed as long as a given condition is true.

Syntax:

---

while logical expression

sequence of statements

end

---

where

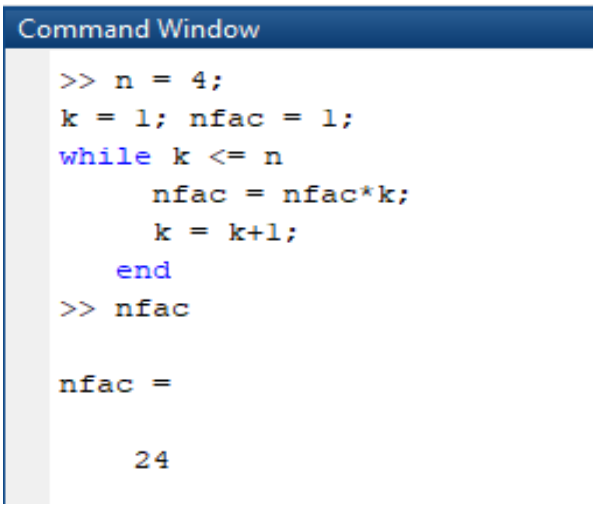
- *logical expression* is an expression whose result can be true or false;
- *sequence of statements* is the processing to be performed while logical expression is true.

Interpretation:

- As long as the logical expression is true, the sequence of statements processing is executed as a loop.

- When logical expression becomes false, we move on to the instruction immediately following the loop end instruction (*end*).

Example:



4 Ventilated choice, the switch statement

An alternative to using a sequence of conditioned statements to make a cascading choice exists. This is the switch statement.

Syntax:

---

switch var

case cst1,

instruction sequence 1

case cst2,

instruction sequence 2

...

case cstN,

---

instruction sequence N

otherwise

default instruction sequence

end

where

- *var* is a numeric variable or a string variable;
- *cst1, ..., cstN,* are numeric constants or string constants;
- *instruction sequence i* is the sequence of instructions to be executed if the contents of the variable var are equal to the constant csti (var= =csti).

Interpretation:

- If the variable var is equal to one of the constants cst1, ..., cstN, (for example csti) then the corresponding sequence of instructions (here sequence of instructions i) is executed.
- The program then resumes at the first instruction following the keyword end.
- If the variable var is not equal to any of the constants the *default sequence of instructions* is executed.

Note:

- The variable var must be of the same type as the constants cst1, ..., cstN.
- It is not necessary to provide a default case (although it is preferable).
    - If there is no default case and if the variable var is not equal to any of the constants, then the program continues at the first instruction following the keyword end.
- It is possible to group several << cases >> if the sequence of instructions to be executed is the same for these different cases. The syntax is then,

case{ cstk , cstl , ...}

common sequence of instructions

Example:

```
Command Window
>> rep = input('Your answer (yes, no, do not know) :', 's');

switch rep
case {'yes','y'},
disp('well done ...');
case {'no','n'}
disp('lost ...');
case 'do not know'
disp('it is easy ...');
end
Your answer (yes, no, do not know) :y
well done ...
>>
```

5 Nested Control Structures

Control structure blocks can include nested control structures.

5.1 Nested Conditional Structures

It is possible to nest conditioned instruction sequences (in the sense that the conditioned instruction sequence contains conditioned instruction sequences). For better readability, it is recommended to use indentations to highlight the nesting of conditioned instruction sequences. It is possible to make a cascading choice:

Syntax:

If logical expression 1

instruction sequence 1

elseif logical expression 2

instruction sequence 2

...

elseif logical expression N

instruction sequence N

else

default instruction sequence

end

Interpretation:

If logical expression 1 is true, instruction sequence 1 is executed and the program then resumes at the first instruction following the keyword end, otherwise if logical expression 2 is true, instruction sequence 2 is executed and the program then resumes at the first instruction following the keyword end, etc. If none of the logical expressions 1 to N are true then the default sequence of statements is executed.

A cascade choice is frequently used when initializing data. For example, we initialize a matrix A based on the value of a variable numex (example number) as follows:

```
if numex == 1
A = ones(n);
 elseif numex == 2
        A = magic(n);
elseif numex == 3 | numex == 4
        A = rand(n);
else
        error('numero d''exemple non prevu ...');
end
```

5.2 Nested Loop

Nested loops are control structures that involve placing a loop inside another loop. They are widely used to handle complex situations and to allow greater flexibility in managing the flow of execution of a program. Nested loops can be used to handle complex repetition patterns or to iterate over multidimensional data sets.

Syntax nested for Loop

```
for i = 1:n % Outer loop
     for j = 1:m % Inner loop
       instruction sequence
     end
end
```

Syntax nested while Loop

```
i = 1; % Initialize outer loop variable
while i <= n % Outer loop
    j = 1; % Initialize inner loop variable
    while j <= m % Inner loop
        instruction sequence
        j = j + 1; % Update inner loop variable
    end
    i = i + 1; % Update outer loop variable
end
```

Syntax Mixed Nested Loops

```
for i = 1:n % Outer loop
  j = 1; % Initialize inner loop variable
  while j <= m % Inner loop
      instruction sequence
      j = j + 1; % Update inner loop variable
  end
end
```

6 Interrupting a control loop

6.1 break

It is possible to cause a premature exit from a control loop. The ***break*** statement is used to exit a for loop or a while loop. Execution then continues sequentially from the statement following the ***end*** keyword, which closes the loop.

In the case of nested loops, only the execution of the inner loop containing the break statement is interrupted.

Example

Command Window

```
>> % Initialize a variable
sum = 0;

% Loop through numbers from 1 to 10
for i = 1:10
    % Add the current number to the sum
    sum = sum + i;

    % If the sum exceeds 20, break out of the loop
    if sum > 20
        disp('Sum exceeded 20, exiting loop.');
        break; % Exit the loop
    end
end
disp(['Final sum: ', num2str(sum)]);
Sum exceeded 20, exiting loop.
Final sum: 21
>>
```

Explanation:

- The loop iterates over numbers from 1 to 10.
- It accumulates the sum of these numbers.

- If the sum exceeds 20, the break statement is triggered, exiting the loop.
- Finally, it displays the sum up to that point.

6.2 return

The ***return*** statement causes a return to the calling program (or to the keyboard). The statements following the return are therefore not executed. The return statement is often used in conjunction with a conditional statement, for example, to test in the body of a function whether the input parameters have the expected values.

Example

```
Command Window
>> % Define an array of numbers
numbers = [3, -1, 4, -2, 5, 6];
target = 5; % The number we're looking for
found = false; % Flag to indicate if the target was found

% Loop through the numbers
for i = 1:length(numbers)
    if numbers(i) == target
        fprintf('Found the target number %d at index %d.\n', target, i);
        found = true; % Update the flag
        return; % Exit the loop (and script)
    end
end

% If we reach here, the target was not found
if ~found
    disp('Target number not found in the array.');
end
Found the target number 5 at index 5.
fx >>
```

Example explanation:

- The script defines an array number and a target number to search for.
- A for loop iterates through each element in the numbers array.
- When the target number is found, it prints a message with the index and sets the found flag to true.

- The return statement exits the script, so any code after that will not execute.
- If the loop completes without finding the target, a message indicates that the target was not found.

6.3 Error

The ***error*** statement is used to stop a program and display an error message. The syntax is ***error('error message')***.

Example:

Command Window

```
>> % Define an array of numbers
numbers = [3, 5, -1, 4, 2];

% Loop through the numbers
for i = 1:length(numbers)
    if numbers(i) < 0
        error('Negative number found: %d at index %d', numbers(i), i);
    end
end

disp('All numbers are non-negative.');
Negative number found: -1 at index 3

fx >>
```

Example explanation:

- The script defines an array of numbers.
- A for loop iterates through each element of the array.
- If a negative number is found, the error statement is triggered, displaying a message that includes the negative number and its index in the array.
- If no negative numbers are found, a message indicating that all numbers are non-negative is displayed.

Running the Script:

- If you run this script, it will terminate and display an error message when it encounters a negative number (-1).
- If you modify the numbers array to remove the negative value, it will print "All numbers are non-negative."

6.4 warning

The ***warning*** statement is used to display a warning message without suspending program execution. The syntax is ***warning('warning message').*** You can tell MATLAB not to display warning messages for a program by typing ***warning off*** in the command window. You can restore the display by typing ***warning on.***

Example:

```
Command Window
>> % Define an array of numbers
numbers = [3, 5, -1, 4, -2, 2];

% Loop through the numbers
for i = 1:length(numbers)
    if numbers(i) < 0
        warning('Negative number found: %d at index %d', numbers(i), i);
    end
end

disp('Loop completed. Check warnings for negative numbers.');
Warning: Negative number found: -1 at index 3
Warning: Negative number found: -2 at index 5
Loop completed. Check warnings for negative numbers.
fx >>
```

Example Explanation:

- The script defines an array number that includes some negative values.
- A for loop iterates through each element of the array.

- If a negative number is found, a warning is issued using the warning statement, which includes the negative number and its index.
- After the loop completes, a message is displayed indicating that the loop has finished.

Running the Script:

- When you run this script, it will display warnings for each negative number found in the array.
- It will continue executing after issuing the warnings and display the final message. This allows you to identify issues without interrupting the flow of the program.

6.5 pause

The ***pause*** command interrupts program execution. Normal execution resumes as soon as the user presses a key on the keyboard. The ***pause(n)*** instruction suspends program execution for n seconds.

Example:

```
Command Window
>> % Display a message
disp('Starting the countdown...');

% Countdown from 5 to 1
for i = 5:-1:1
    fprintf('%d\n', i);
    pause(1);  % Pause for 1 second
end

% Final message
disp('Countdown complete!');
Starting the countdown...
5
4
3
2
1
Countdown complete!
fx >>
```

Example Explanation:

- The script begins by displaying a starting message.
- It then enters a loop that counts down from 5 to 1.
- Inside the loop, it displays the current number and pauses for 1 second using pause(1).
- After the countdown completes, it displays a final message.

When you run this code, you'll see the countdown numbers displayed one by one with a 1-second pause between them.

Chapter 6: Function file

1 Introduction

2 Definition of a function

3 Difference between script and function

4 Structure of a MATLAB function

1 Introduction

There are many predefined functions in MATLAB, but there will inevitably come a time when you want to use a function that is not defined. Fortunately, it is possible to define your functions and use them exactly like the pre-existing functions.

2 Definition of a function

A function is a self-contained block of code that performs a specific task. In programming, functions take inputs, perform operations on those inputs, and return an output. It helps structure code logically and make it reusable.

3 Difference between script and function

In MATLAB, script and functions are two types of files used to execute blocks of code, but they have different characteristics and uses.

- A script is a file containing a series of MATLAB statements that run in the current workspace. Scripts do not take arguments and do not return values.
- A function is a file that can take input arguments, perform calculations, and return values. Functions have their own workspace, which means that they cannot directly access variables in the main workspace unless they are declared as global variables.

4 Structure of a MATLAB function

According to the rules of MATLAB the syntax used for writing a function is as follows:

syntax

```
function [vars1, ..., varsm] = name_function(vare_1, ..., varen)
     statement sequence
end
```

where

- *vars1*, ..., *varsm*: are the output variables of the function;
- *vare1*, ..., *varen*: are the input variables of the function;
- *statement sequence:* is the body of the function.

The file must begin with the keyword ***function***. The output variables of the function follow in brackets, the symbol =, the name of the function and finally the input variables in parentheses. If the function has only one output variable, the brackets are unnecessary. The function named func must be saved in a file named func.m otherwise this function will not be << visible >> by MATLAB.

Example 1: Function with One Output.

A function that adds two numbers

```
function s = addition(a, b)

    s = a + b;

end
```

Call the function from the command line: s = addition(10, 20)

Example 2: Function with Multiple Outputs

Define a function in a file named stat.m that returns the mean and standard deviation of an input vector.

```
function [m,s] = stat(x)

   n = length(x);

   m = sum(x)/n;

   s = sqrt(sum((x-m).^2/n));

end
```

Call the function from the command line.

values = [12.7, 45.4, 98.9, 26.6, 53.1];

[ave,stdev] = stat(values)

Example 3: Function Without Output

Define a function in a file named `plotData.m` that plots inputs using custom parameters.

```
function plotData(Xdata,Ydata)
   plot(Xdata,Ydata,Color="black",LineStyle="-.")
end
```

Call the function from the command line.

Xdata = 1:100;

Ydata = sin(pi/20*Xdata);

plotData(Xdata,Ydata)

Example 2: Multiple Functions in a Function File

Define two functions in a file named `stat2.m`, where the first function calls the second.

```
function [m,s] = stat2(x)
   n = length(x);
   m = avg(x,n);
   s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
   m = sum(x)/n;
end
```

Function avg is a *local function*. Local functions are only available to other functions within the same file.

Call function stat2 from the command line.

values = [12.7, 45.4, 98.9, 26.6, 53.1];

[ave,stdev] = stat2(values)

Chapter 7: Graphics

1 Introduction

2 2D

3 3D graphics

4 specific graphs

5 Graphical User Interfaces in MATLAB

1 Introduction

MATLAB provides a powerful set of graphical tools that make it easy to plot data and visualize mathematical functions using just a few commands. This simplicity encourages frequent use of graphing to better understand mathematical concepts, as visualizing equations can be a fun and effective way to learn. Matlab offers a wide range of functions for manipulating graphical objects, and in this chapter, we will explore basic 2D graphing techniques, 3D graphics, surface plots, and interactive user interfaces (GUIs).

2 2D Graphics

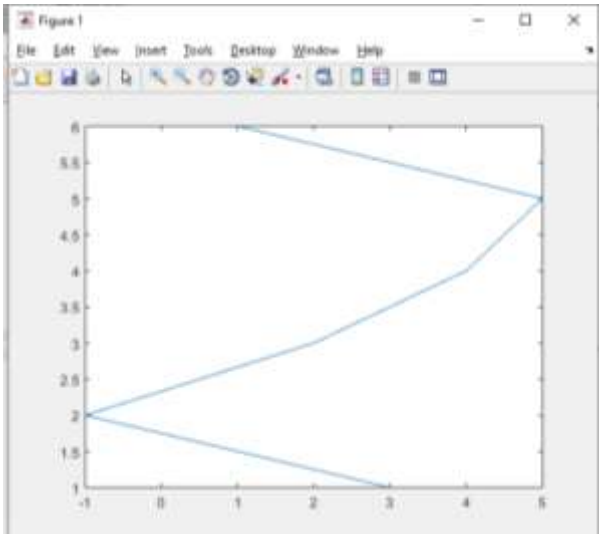
2.1 Creating simple plots

The basic MATLAB graphing procedure is to Take a vector of x-coordinates, x = (x_1,..., x_n), and a vector of y-coordinates, y = (y_1,..., y_n), finding the points (xi, yi) with i = 1, 2,..., n, and joining them by straight lines. x and y are both row arrays or column arrays of the same length.

plot(x,y) is the MATLAB command to plot a graph. The image in Figure below is the result of the vectors x = (3, −1, 2, 4, 5, 1)and y = (1, 2, 3, 4, 5, 6).

Example 1:

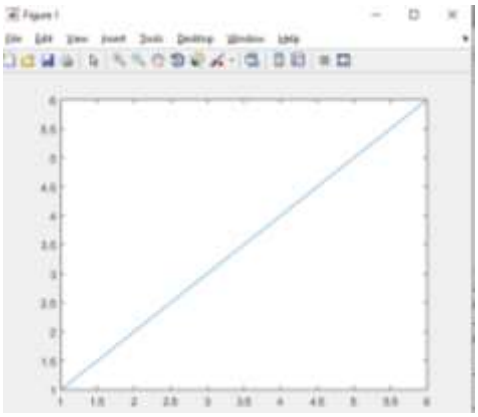
```
Command Window
>> x= [3, -1, 2, 4, 5, 1]; y= [1, 2, 3, 4, 5, 6];
>> plot(x,y)
>>
```



Note: The plot functions have different forms depending on the input arguments. If y is a vector plot(y) produces a piecewise linear graph of the elements of y versus the index of the elements of y. If we specify two vectors, as mentioned above, plot(x,y) produces a graph of y versus x.

Command Window

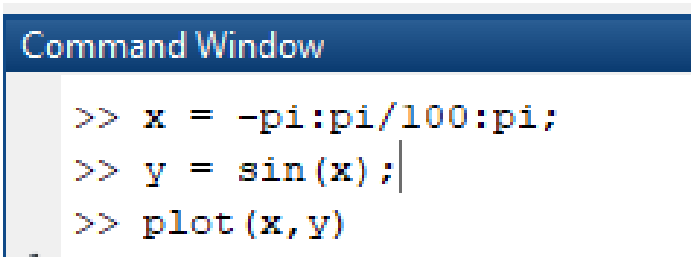
```
>> x= [3, -1, 2, 4, 5, 1]; y= [1, 2, 3, 4, 5, 6];
>> plot(y)
>>
```

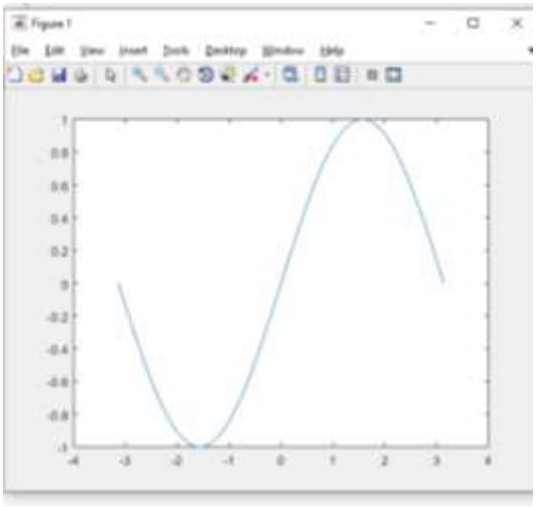


Example 2:

Notes:

- -pi:pi/100:pi yields a vector that
  - starts at -π,
  - takes steps (or increments) of π/100,
  - stops when π is reached.
- If you omit the increment, MATLAB automatically increments by 1.

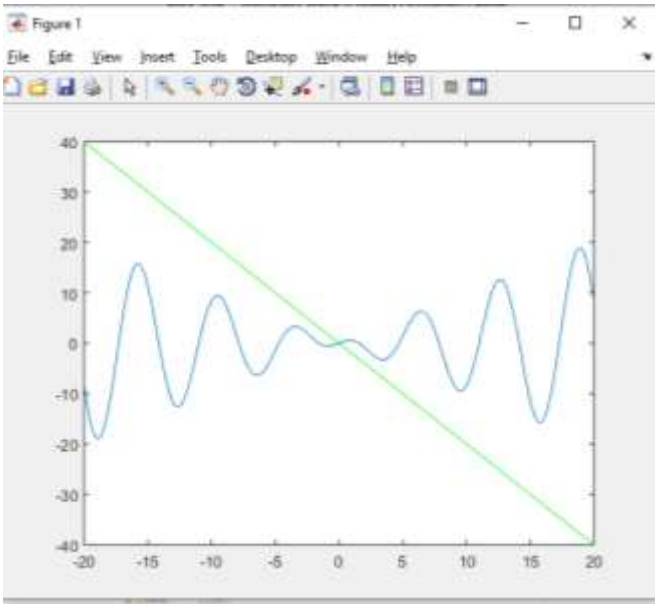




2.2 Multiple data sets in one plot: hold on hold off

With each new plot command, the figure is replaced. To keep multiple curves, you must allow graph superposition using the hold on command. The following plots will be superimposed until hold off is deactivated or the window is closed.

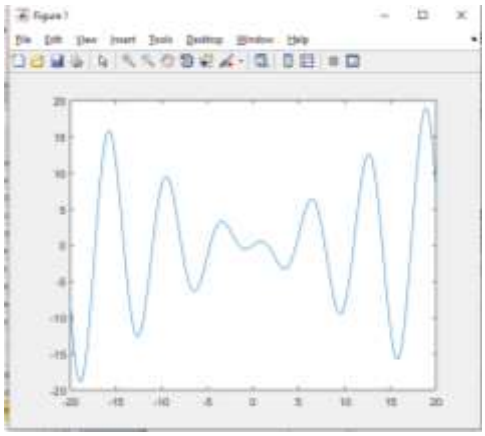
```
Command Window
>> x = linspace(-20,20,1000);
>> y3 = x.*cos(x);
>> x = linspace(-20,20,1000);
>> y = x.*cos(x);
>> plot(x,y)
>> hold on
>> y2 = -2*x;
>> plot(x,y2,'g')
```

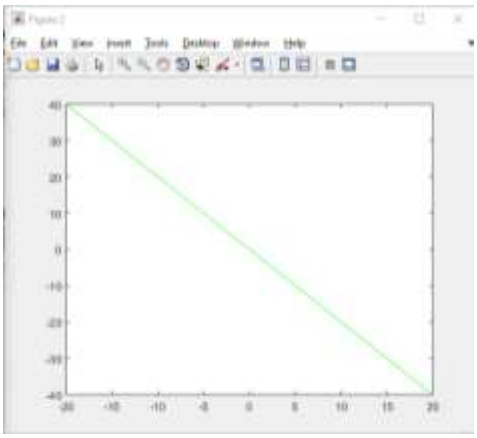


2.3 **Multiple curves on multiple windows**

It is also possible to plot multiple curves on multiple windows. To do this, a new window (figure graphic object) must be invoked before calling the corresponding plot function.

```
Command Window
>> plot(x,y)
>> figure(2)
>> plot(x,y2,'g')
>>
```

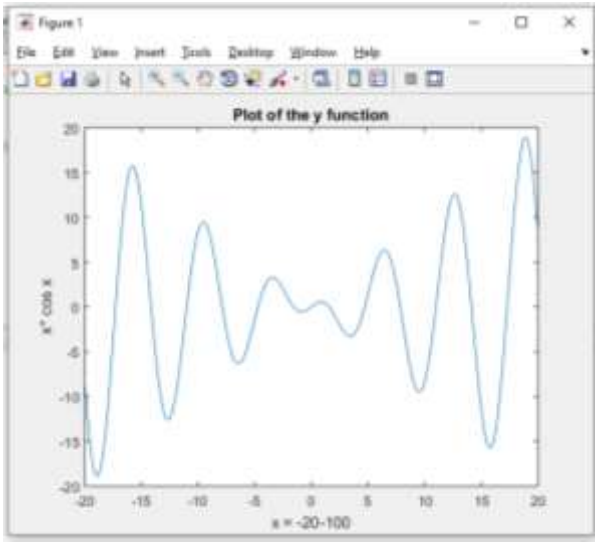




2.4 **Adding titles, axis labels, annotations, and colour**

In MATLAB, use xlabel, ylabel, and title to add axis labels and titles.

```
Command Window
>> xlabel('x = -20-100')
>> ylabel('x* cos x')
>> title('Plot of the y function')
>>
```



The colour of a single curve is, by default, blue, but other colours are possible. The desired colour is indicated by a third argument. For example, green is selected by plot(x,y,'g'). Note the single quotes, ' ', around g.

Moreover, it is possible to specify line styles, colours, and markers (e.g., circles, plus signs, . . .) using the plot command:

plot(x,y,'style_color_marker')

Where style_color_marker is a triplet of values from Table 1. To find additional information, type help plot or doc plot.

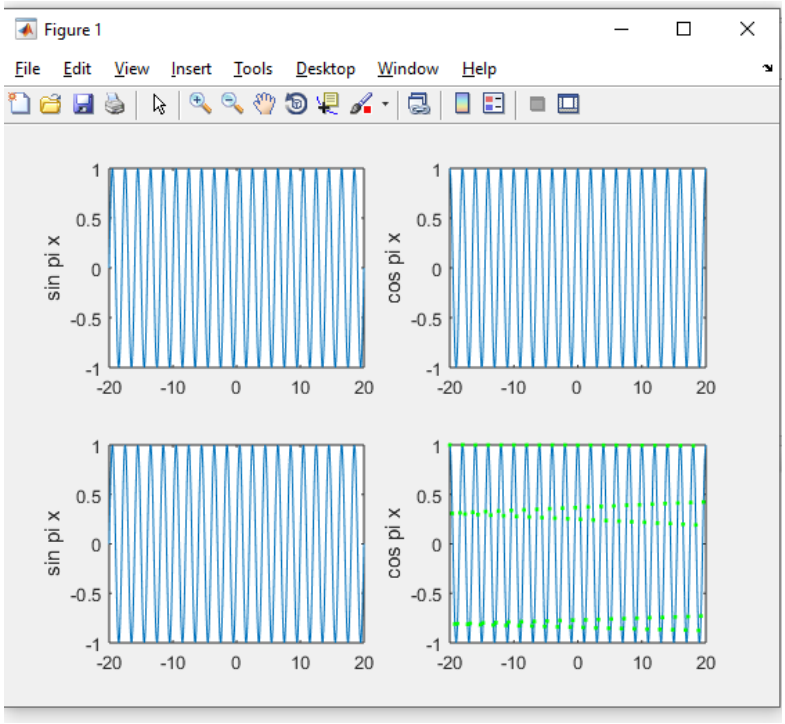
Symbol	Color	Symbol	Line Style	Symbol	Marker
K	Black	-	Solid	+	Plus sign
r	Red	--	Dashed	o	Circle
b	Blue	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
c	Cyan	None	No line	x	cross
m	Magenta			s	Square
y	Yellow			d	Diamond

Table 1: Plot attributes

2.5 **Displaying multiple graphs side by side: subplot**

A graph window can be divided into sub-windows according to a table of dimensions (m*n). One or more curves can then be displayed in each sub-window:

```
Command Window
>> subplot(2,2,1), plot(x,sin(pi*x))
>> ylabel('sin pi x')
>> subplot(2,2,2), plot(x,cos(pi*x))
>> ylabel('cos pi x')
>> subplot(2,2,3), plot(x,sin(pi*x))
>> ylabel('sin pi x')
>> subplot(2,2,4), plot(x,cos(pi*x)),
>> hold on
>> plot(x(1:10:end),cos(pi*x(1:10:end)),'g.')
>> ylabel('cos pi x')
fx >>
```



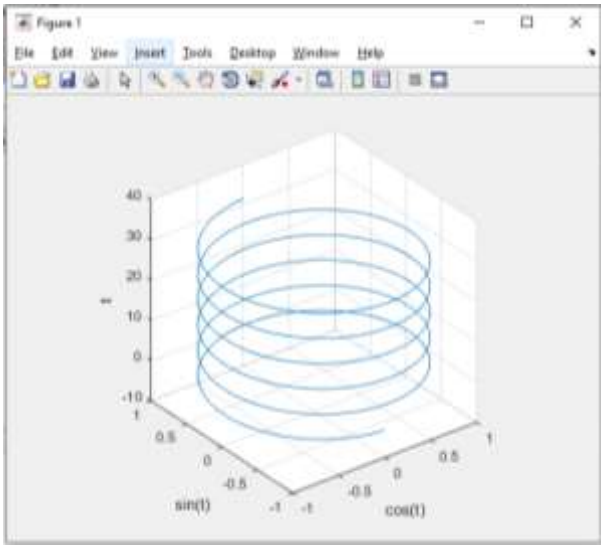
3 3D graphics

3.1 Drawing curves in space

The example below shows how to plot curves in space. The function takes 3 vectors of the same size as arguments. Its operation is similar to that of the plot. It displays in a 3-dimensional axis system the triplets [x(i),y(i),z(i)].

Command Window

```
>> t = -pi:pi/50:10*pi;
 plot3(sin(t),cos(t),t)
 grid on
axis square
xlabel('cos(t)'), ylabel('sin(t)'), zlabel('t')
>>
```



3.2 Mesh representation in the (x,y) plane

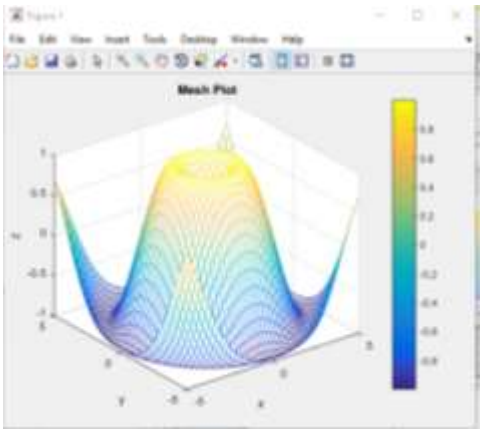
The example below illustrates how to use the mesh and surf functions in MATLAB, along with the meshgrid function to generate the necessary matrices.

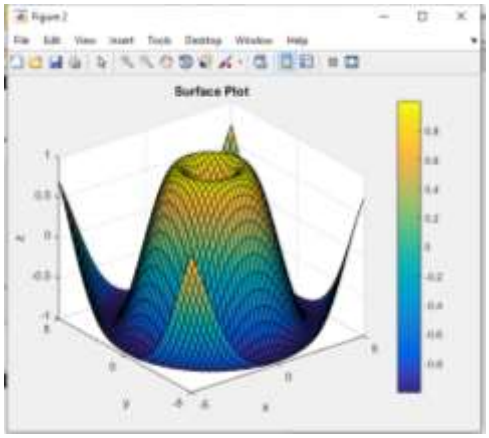
These steps illustrate how you can generate and visualize 3D data using MATLAB's mesh and surf functions. The colorbar provides a visual representation of the values, enhancing the interpretability of the plots.

1. **Define the vectors for** x **and** y:
   - We create two vectors, `x` and `y`, spanning from -5 to 5 with 50 points each.
2. **Generate the grid matrices**:
   - Using meshgrid, we create the matrices X and Y which represent all combinations of the values in x and y.
3. **Define the function**:
   - Z = sin(sqrt(X.^2 + Y.^2)) defines the height of the surface at each grid point.
4. **Create a mesh plot**:
   - The mesh function creates a 3D wireframe plot of the surface. The colorbar function adds a colour scale to the plot.
5. **Create a surface plot**:
   - The surf function creates a 3D surface plot. Like the mesh plot, a colour scale is added using colorbar.

Command Window

```
>> x = linspace(-5, 5, 50); y = linspace(-5, 5, 50);
[X, Y] = meshgrid(x, y);
Z = sin(sqrt(X.^2 + Y.^2));
figure;
mesh(X, Y, Z);
title('Mesh Plot');
xlabel('x');
ylabel('y');
zlabel('z');
colorbar; % Add a colorbar for color scale
grid on;
figure;
surf(X, Y, Z);
title('Surface Plot');
xlabel('x');
ylabel('y');
zlabel('z');
colorbar; % Add a colorbar for color scale
grid on;
fx >>
```





3.3 **Drawing contour curves**

The example below creates contour plots that display constant z curves on the x-y plane, illustrating the use of the contour function.

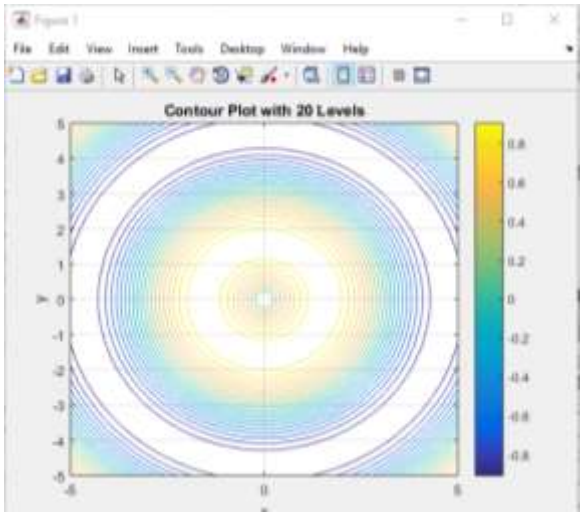
These steps show how to generate and visualize 2D contour plots using MATLAB's contour function. The colorbar provides a visual representation of the values, enhancing the interpretability of the plots.

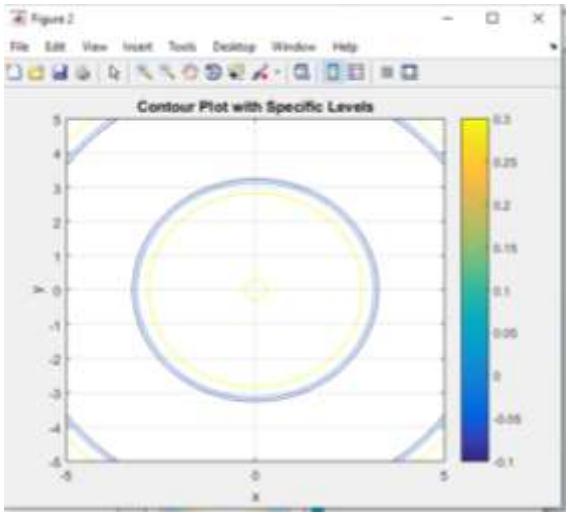
- **Define the vectors for x and y:**
  - We create two vectors, `x` and `y`, spanning from -5 to 5 with 50 points each.
- **Generate the grid matrices**:
  - Using meshgrid, we create the matrices X and Y which represent all combinations of the values in x and y.
- **Define the function**:
  - Z = sin(sqrt(X.^2 + Y.^2)) defines the height of the surface at each grid point.
- **Create a contour plot with 20 levels**:
  - The contour function is used to create a 2D contour plot with 20 contour levels, representing different values of z on the x-y plane. The colorbar function adds a colour scale to the plot.

- **Create a contour plot with specific levels**:
    - The contour function is used again to create a 2D contour plot, but this time with specified contour levels [−0.1,0,0.3][-0.1, 0, 0.3], representing the curves where z=−0.1, z = 0, and z = 0.3.

Command Window

```
>> x = linspace(-5, 5, 50);
y = linspace(-5, 5, 50);
[X, Y] = meshgrid(x, y);
Z = sin(sqrt(X.^2 + Y.^2));
figure;
contour(X, Y, Z, 20); % 20 contour levels
title('Contour Plot with 20 Levels');
xlabel('x');
ylabel('y');
colorbar; % Add a colorbar for color scale
grid on;
figure;
contour(X, Y, Z, [-0.1 0 0.3]); % Specific contour levels
title('Contour Plot with Specific Levels');
xlabel('x');
ylabel('y');
colorbar; % Add a colorbar for color scale
grid on;
fx >>
```





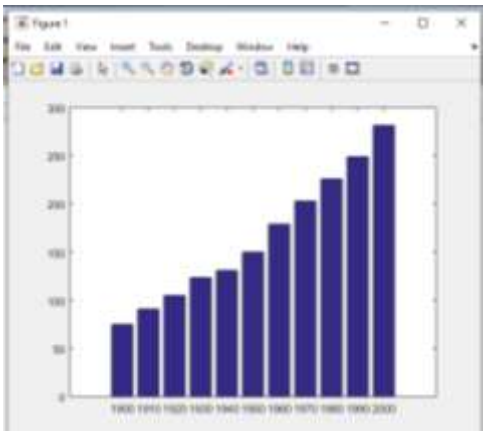
4 specific graphs

MATLAB allows to draw specific graphs

• Bars: Bar

bar(x,y) draws the bars at the locations specified by x.

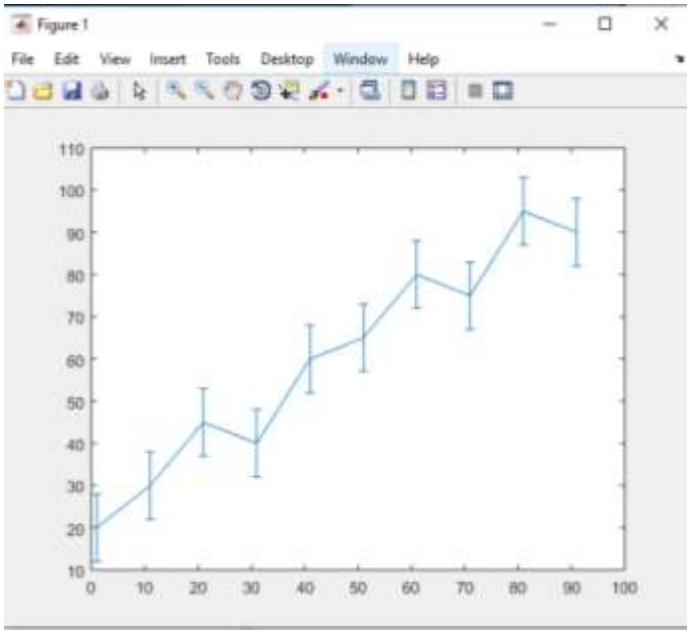
```
Command Window
>> x = 1900:10:2000;
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(x,y)
>>
```



• Error bar: errorbar

errorbar(x,y,err) plots y versus x and draws a vertical error bar at each data point.

```
Command Window
>> x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = 8*ones(size(y));
errorbar(x,y,err)
>>
```



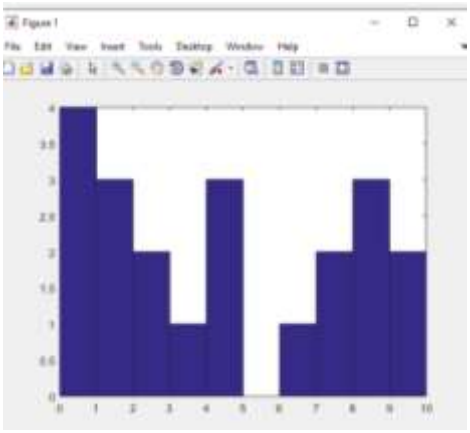
• Histogram: hist

hist(x) creates a histogram bar chart of the elements in vector x. The elements in x are sorted into 10 equally spaced bins along the x-axis between the minimum and maximum values of x. hist displays bins as rectangles, such that the height of each rectangle indicates the number of elements in the bin.

If the input is a multi-column array, hist creates histograms for each column of x and overlays them onto a single plot.

If the input is of data type categorical, each bin is a category of x.

```
Command Window
>> x = [0 2 9 2 5 8 7 3 1 9 4 3 5 8 10 0 1 2 9 5 10]
hist(x)
>>
```

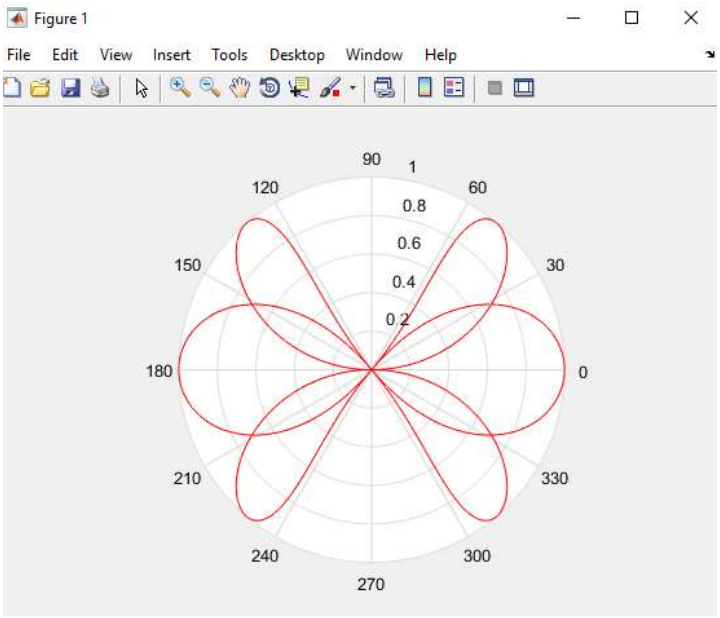


• Polar coordinates: polar

polar(theta,rho,LineSpec) specifies the line style, marker symbol, and colour for the lines drawn in the polar plot.

Command Window

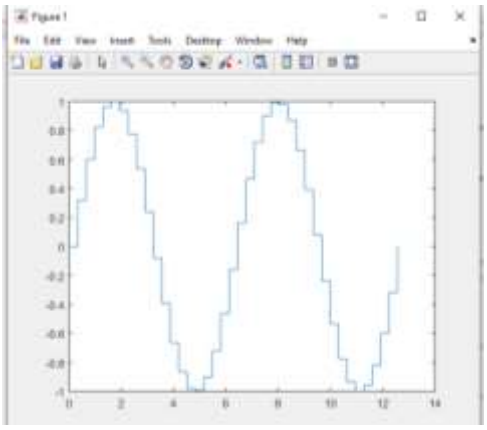
```
>>  x=linspace(0,1,1000);
polar(sin(8.*x.*pi), cos(12.*x.*pi),'--r'
>>
```



• Stair step: stairs

stairs(X,Y) plots the elements in Y at the locations specified by X. The inputs X and Y must be vectors or matrices of the same size. Additionally, X can be a row or column vector and Y must be a matrix with length(X) rows.

```
Command Window
>> X = linspace(0,4*pi,40);
Y = sin(X);
stairs(X,Y)
>>
```

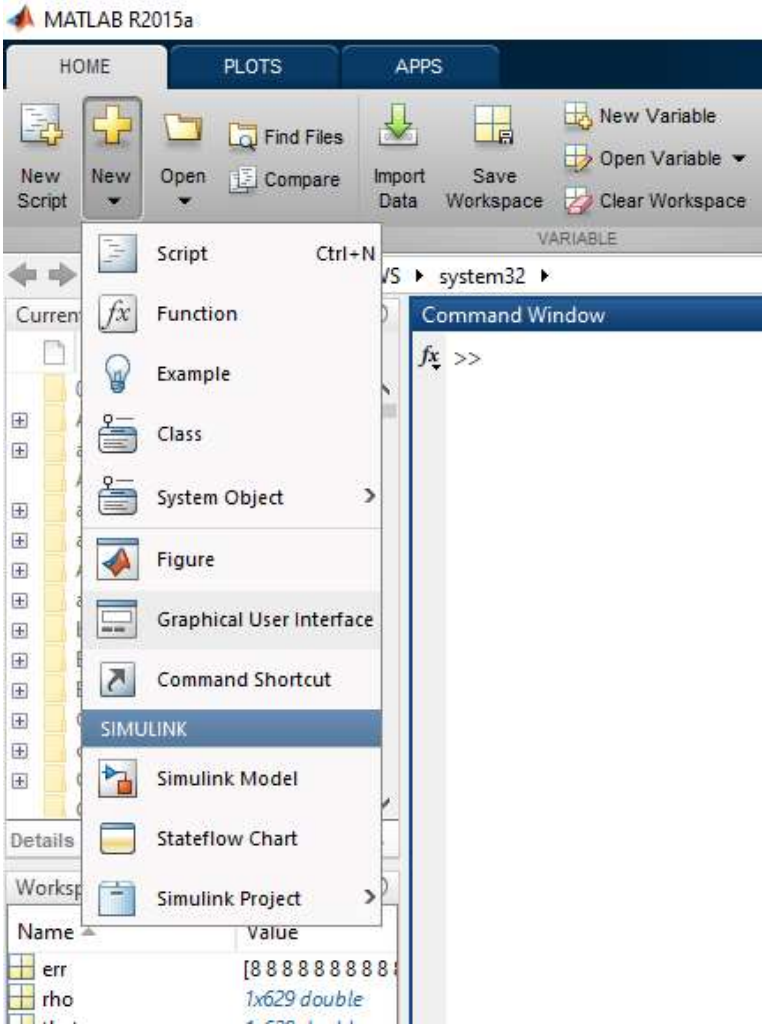


5 Graphical User Interfaces in MATLAB

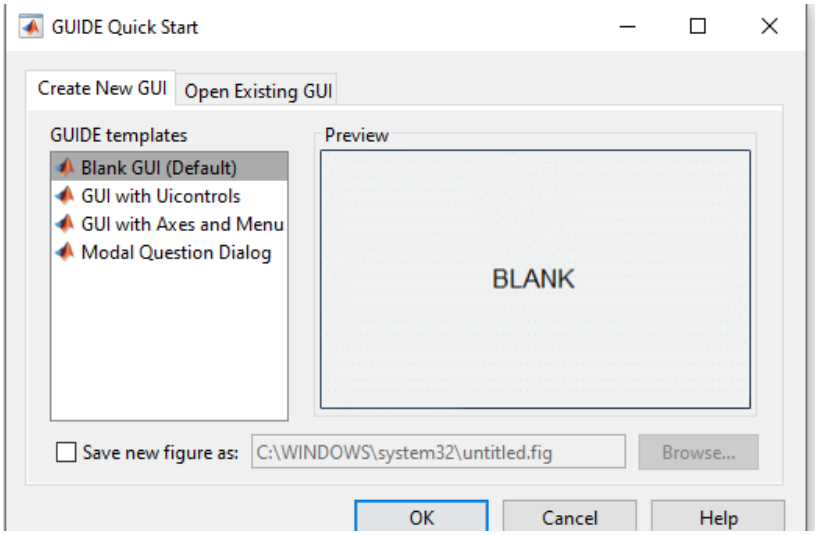
Graphical user interfaces (GUI) are objects that allow users to have inputs using graphical interfaces, such as: buttons, radio buttons, context menus, etc. GUIs allow controlling software applications with point-and-click commands. In MATLAB, there are two basic ways to create GUIs:

1. Create a MATLAB GUI interactively: Use the built-in GUI Development Environment (GUI). GUI allows the user to graphically present the GUI and MATLAB automatically generates the corresponding code.
2. Create a MATLAB GUI programmatically: To be able to understand and modify this code, it is important to understand the underlying programming concepts.

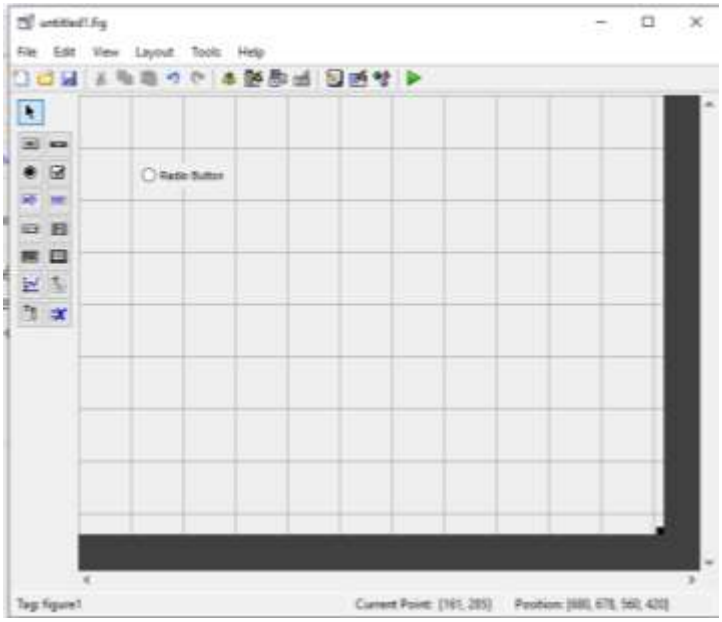
In our case, we focus on the first method. The GUI is opened either by typing gui in the MATLAB Command Window or by choosing new-> Graphical User Interface.



The following window appears.



Object placement is done by selecting from a toolbox. Their placement and sizing are done using the mouse. In the Figure below, the GUI is created and the toolbox is on the left.



Chapter 8: Toolbox

1 Introduction

2 Definition

3 The existing toolboxes

1 Introduction

MATLAB toolboxes extend the core functionality of MATLAB for specific applications. They provide specialized functions and algorithms that facilitate the analysis, modelling, and solving of complex problems. Using them can save time and improve the efficiency of computations and visualizations. This chapter explores the main toolboxes.

2 Definition

Matlab can be enriched by adding toolboxes which are sets of additional functions, profiled for particular applications.

Toolboxes are collections of M files developed for specific application domains (Signal Processing Toolbox, System Identification Toolbox, Control System Toolbox, u-Synthesis and Analysis Toolbox, Robust Control Toolbox, Optimization Toolbox, Neural Network Toolbox, Spline Toolbox, Chemometrics Toolbox, Fuzzy Logic Toolbox, etc.)

3 The existing toolboxes

The toolboxes existing from version 5.3 are:

- Statistics Toolbox, Signal Processing Toolbox, Image Processing Toolbox, Fuzzy Logic Toolbox, Neural Networks Toolbox, Spline Toolbox, Wavelet Toolbox, Mapping Toolbox,
- Control System Toolbox, Optimization Toolbox, Robust Control Toolbox, System Identification Toolbox, Higher-Order Spectral Analysis Toolbox, DSP Blockset, Frequency Domain, Mu Analysis and Synthesis Toolbox, Power System Blockset, Data Acquisition Toolbox,

- Database Toolbox, Financial Toolbox, Communications Toolbox, MATLAB Web Server,
- Symbolic Math Toolbox, Partial Differential Equations (PDE) Toolbox,

3.1 Statistics Toolbox

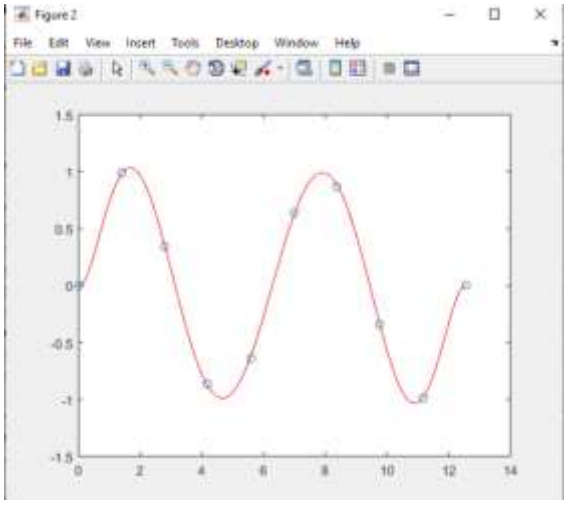
The functions of the statistics toolbox can be classified as follows:

- The functions (distributions) of probability density and cumulative pdf: beta, binomial, Chi2, exponential, F, gamma, normal, Poisson, Rayleigh, T, Uniform, Weibull,…
- The inverse functions of all these functions are also available: betainv, binoinv, chi2inv, gaminv, norminv, poissinv, raylinv, weibinv,…
- The generators of random numbers distributed according to these distributions: betarnd, binornd, chi2rnd, frnd, gamrnd, lognrnd, normrnd, poissrnd, unifrnd, weibrnd,…
- Statistics of all these distributions (mean, variance, …): betastat, binostat, chi2stat, gamstat, poisstat, weibstat, …
- Descriptive statistics: mean, median, geomean, harmmean, std, var, skewness, kurtosis, Kurtosis, iqr (interquartile range), corrcoef, cov,…

Example1 : Polynomial curve fitting:

```
%Generate 10 points equally spaced along a sine curve in the interval
[0,4*pi].
        x = linspace(0,4*pi,10);
        y = sin(x);
        %Use polyfit to fit a 7th-degree polynomial to the points.
        p = polyfit(x,y,7);
        %Evaluate the polynomial on a finer grid and plot the results.
        x1 = linspace(0,4*pi);
        y1 = polyval(p,x1);
```

```
plot(x,y,'o')
hold on
plot(x1,y1,'r')
```



.

Bibliography

1. Houcque, D. (2005). Introduction to Matlab for engineering students. Northwestern University, 1.

2. Ariba, Y., & Cadieux, J. (2015). 'Manuel MATLAB. Consulté le: 21 septembre 2024. [En ligne]. Disponible sur: https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-matlab.pdf

3. Quentin Glorieux. Outils Mathématiques et utilisation de Matlab (Cours 2013-2014), Université Pierre et Marie Curie –ParisV, Consulté le: 21 septembre 2024. [En ligne]. Disponible sur: https://www.lkb.upmc.fr/quantumoptics/wp-content/uploads/sites/23/2015/12/Matlab2013.pdf

4. Jonas KOKO. COURS DE MATLAB. 2020 Institut Supérieur d'Informatique, de Modélisation et de leurs Applications Campus des Cézeaux. Consulté le: 22 décembre 2024. [En ligne]. Disponible sur: https://perso.isima.fr/~jokoko/doc/polymatlab.pdf

5. « Hahn et Valentine - 2010 - Essential MATLAB for engineers and scientists.pdf ». Consulté le: 22 décembre 2024. [En ligne]. Disponible sur: https://faculty.ksu.edu.sa/sites/default/files/Essential+MATLAB+for+Engineers+and+Scientists+Fourth+Edition.pdf

6. Hahn et Valentine - 2010 - Essential MATLAB for engineers and scientists.pdf ». Consulté le: 22 décembre 2024. [En ligne]. Disponible sur: https://faculty.ksu.edu.sa/sites/default/files/Essential+MATLAB+for+Engineers+and+Scientists+Fourth+Edition.pdf

7. « MATLAB EXPO ». Consulté le: 22 décembre 2024. [En ligne]. Disponible sur: https://www.matlabexpo.com/