PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION & SCIENTIFIC RESEARCH

Handout of the course on Computer Science 3 intended for second-year State Engineering students in Mechanical Engineering, Science and Technology

By Dr.BELLA MOURAD

Teacher at the department of Mechanical and Electromechanical Engineering

Institute of Scienceand Technology

Abdel Hafid Boussouf university center, Mila

2025/2026

Table of Contents

Chapter 1: Presentation of a Scientific Programming Environment

1.1 Introduction

Scientific programming environments are essential tools for engineers, scientists, and researchers. They provide powerful platforms for performing mathematical computations, data visualization, numerical simulations, and system modeling. Such environments combine programming languages with advanced mathematical libraries, making them indispensable in solving real-world engineering and scientific problems.

Among the most widely used environments are **MATLAB**, **Python**, and **GNU Octave**. Each offers a set of tools to perform high-level programming, numerical computation, and data analysis efficiently.

In this course, we will focus mainly on **MATLAB (Matrix Laboratory)** — a leading scientific computing environment widely used in academic and industrial research for modeling, simulation, and algorithm development.

1.2 What is MATLAB?

MATLAB is a high-level language and interactive environment developed by **MathWorks**. It allows users to perform matrix-based computations, design algorithms, visualize data, and create models and applications.

The name MATLAB comes from **"Matrix Laboratory"**, highlighting the fact that matrices and arrays are the core data elements in MATLAB.

It is especially useful in fields such as:

- Mechanical and electrical engineering
- Signal and image processing
- Control systems
- Machine learning and data analysis
- Computational mathematics

1.3 Characteristics of a Scientific Programming Environment

A good scientific programming environment such as MATLAB provides several essential features:

1. **Interactive Interface** – Allows users to type commands and see immediate results.
2. **Mathematical Functions** – Includes built-in functions for algebra, calculus, differential equations, and statistics.
3. **Visualization Tools** – Provides 2D and 3D plotting tools to visualize data and models.
4. **Programming Language** – Supports script and function files for automation and modular programming.

5. **Toolboxes** – Offers specialized packages for specific domains like control systems, optimization, and signal processing.
6. **Simulation Environment** – Integrates with Simulink for graphical modeling and simulation of dynamic systems.

1.4 Starting MATLAB

When MATLAB is launched, the main interface appears, composed of several key components:

- **Command Window:**
  The area where you can type commands and execute them immediately. Example:
- >> a = 5;
- >> b = 10;
- >> c = a + b
- c =
- 15
- **Workspace:**
  Displays all current variables stored in memory.
- **Command History:**
  Shows all previously entered commands, allowing quick reuse.
- **Editor/Debugger:**
  Used to create and edit MATLAB script files (.m files).
- **Current Folder Browser:**
  Allows you to navigate through directories and manage your project files.
- **Figure Window:**
  Displays graphical results such as plots and diagrams.

1.5 Basic MATLAB Operations

MATLAB uses a matrix-oriented language, meaning all data are treated as arrays or matrices.

Arithmetic Operations

>> x = 10;

>> y = 3;

>> x + y

ans = 13

>> x / y

ans = 3.3333

Vector and Matrix Definition

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

Element-by-Element Operations

```
>> D = A .* B
```

Transposition and Inverse

```
>> A'
>> inv(A)
```

1.6 Plotting and Visualization

Visualization is one of MATLAB's strongest features. You can plot data easily with simple commands.

Example:

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x, y);
title('Sine Wave');
xlabel('x');
ylabel('sin(x)');
grid on;
```

This will generate a smooth sine wave curve with labeled axes and grid lines.

1.7 Saving and Loading Work

MATLAB allows saving your workspace variables for later use.

Save all variables:

```
save('mydata.mat')
```

Load saved variables:

```
load('mydata.mat')
```

Clear workspace:

```
clear
```

1.8 Comparison with Other Environments

Although MATLAB is one of the most powerful scientific computing tools, it is not the only one available.

Here is a brief comparison:

Feature	MATLAB	Python (NumPy, SciPy)	Octave
License	Commercial	Open-source	Open-source
Speed	Very Fast	Fast (with libraries)	Moderate
Interface	Integrated GUI	Text-based (can use Jupyter)	Similar to MATLAB
Compatibility	Industry Standard	Widely Used	Compatible with MATLAB Scripts

MATLAB remains the preferred choice in academic and industrial engineering due to its robustness, documentation, and specialized toolboxes.

1.9 Advantages of MATLAB

- Easy-to-learn high-level syntax
- Rich visualization capabilities
- Extensive mathematical libraries
- Integration with Simulink for dynamic systems
- Strong support for engineering toolboxes
- Large user community and documentation

1.10 Practical Example

Problem:

Compute and plot the motion of a damped oscillator defined by:

[
x(t) = e^{-0.1t} \cos(2\pi t)
]

Solution:

```
t = 0:0.01:10;
x = exp(-0.1*t).*cos(2*pi*t);
plot(t, x);
xlabel('Time (s)');
ylabel('Displacement');
title('Damped Oscillation');
grid on;
```

This example demonstrates the simplicity and efficiency of MATLAB in solving physical problems related to mechanical systems.

1.11 Exercises

Exercise 1

Open MATLAB and perform the following:

1. Create two variables a and b, assign them values, and compute:
    - (a + b)
    - (a^2 + b^2)
2. Save the results in a file named results.mat.

Exercise 2

Plot the following function:

[

y = e^{-t} \sin(2\pi t)

]

for (t) ranging from 0 to 5.

Exercise 3

Create a 3×3 matrix and:

1. Compute its transpose.
2. Calculate its determinant using det(A).
3. Find its inverse using inv(A).

1.12 Solutions

Solution 1

```
a = 4; b = 6;
sum1 = a + b;
sum2 = a^2 + b^2;
save('results.mat');
```

Solution 2

```
t = 0:0.01:5;
y = exp(-t).*sin(2*pi*t);
plot(t, y);
title('Exponential Sine Function');
xlabel('t'); ylabel('y');
```

Solution 3

```
A = [1 2 3; 4 5 6; 7 8 9];At = A';d = det(A);invA = inv(A);
```

Chapter 2: Script Files and Types of Data and Variables

2.1 Introduction

In MATLAB, all computations are performed using variables and data types.

To automate tasks and organize code efficiently, users write programs called **script files** or **functions**.

This chapter introduces the concept of script files, explains how to define and manipulate variables, and presents the main data types used in MATLAB.

A good understanding of variables and data structures is essential because MATLAB is fundamentally **matrix-oriented**, and almost every operation involves arrays or matrices.

2.2 Script Files

2.2.1 Definition

A **script file** is a simple text file containing a sequence of MATLAB commands.

It is used to execute a set of instructions automatically, rather than typing them one by one in the Command Window.

Script files are saved with the extension **.m** (called **M-files**).

2.2.2 Creating a Script File

To create a script:

1. Open the **MATLAB Editor** (Home → New Script).
2. Write your commands in the editor.
3. Save the file with a meaningful name (e.g., example.m).
4. Run the script by typing its name in the **Command Window** (without .m).

Example:

```
% Example script file: example.m
a = 5;
b = 10;
c = a + b;
disp(['The sum is: ', num2str(c)]);
```

Result:

```
The sum is: 15
```

2.2.3 Comments in Scripts

Comments start with a percent sign % and are ignored by MATLAB during execution.

They are useful for documenting code.

```
% This program calculates the area of a circle
r = 5;
```

```
A = pi * r^2;
disp(['Area = ', num2str(A)]);
```

You can also use **block comments**:

```
%{
This block comment
is ignored by MATLAB.
%}
```

2.2.4 Advantages of Script Files

- Allow repetitive tasks to be automated.
- Make programs easier to read and debug.
- Facilitate sharing and documentation of code.
- Enable step-by-step execution for learning and testing.

2.3 Variables

2.3.1 Definition

A **variable** is a symbolic name used to store a value in memory.

In MATLAB, variables are created automatically when you assign them a value using the = operator.

Example:

```
x = 12;
y = 4.5;
name = 'Student';
```

2.3.2 Naming Rules for Variables

Variable names in MATLAB must follow certain rules:

1. **Must start with a letter.**
   Example: a1 is valid, but 1a is invalid.
2. **May contain letters, digits, or underscores.**
   Example: data_2, Temperature, velocity1
3. **Must not contain special characters**
   such as é, @, #, %, :, ;, (), etc.
4. **Must not be the name of an existing MATLAB function.**
   Example: Avoid using sum, mean, plot as variable names.

Example:

```
log = 5;
clear log   % To restore the MATLAB 'log' function
```

2.3.3 The Command clear

If a variable name conflicts with a MATLAB function, you can delete it using:

clear variable_name

Example:

clear i j % restores i and j as imaginary units

i = sqrt(-1)

2.4 Data Types

MATLAB supports different **data types** (classes) for representing information.

The three most common are:

1. **Real numbers**
2. **Complex numbers**
3. **Character strings**

2.4.1 Real Numbers

A real number can be an integer or a decimal value.

Example:

a = 3;

b = 4.5;

MATLAB automatically recognizes numbers as **double precision** by default.

2.4.2 Complex Numbers

Complex numbers are written in the form

[

z = a + bi

]

where (i) (or (j)) is the imaginary unit.

Example:

z1 = 2 + 3i;

z2 = 4 - 2j;

Useful functions:

real(z) % real part

imag(z) % imaginary part

abs(z) % magnitude

angle(z) % phase (radians)

conj(z) % complex conjugate

2.4.3 Character Strings

A **string** is a sequence of characters enclosed in single quotes ' '.

Example:

name = 'Mechanical Engineering';

You can concatenate strings using square brackets:

fullName = ['Department of ', name];

Display:

disp(fullName)

2.5 The Workspace

The **Workspace** shows all active variables in memory.

You can view them in the Command Window with:

who

whos

To clear all variables:

clear all

To clear the screen:

clc

2.6 Input and Output

a) Input from the User

x = input('Enter a value for x: ');

b) Displaying Output

disp(['The value of x is ', num2str(x)]);

fprintf('The result is %.2f\n', x);

2.7 Practical Example

Program: Calculate the total resistance of three resistors connected in parallel.

[

R_t = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}

]

Script:

% Program to compute total resistance

R1 = input('Enter R1: ');

R2 = input('Enter R2: ');

R3 = input('Enter R3: ');

Rt = 1 / (1/R1 + 1/R2 + 1/R3);

fprintf('Total resistance = %.2f Ohms\n', Rt);

Execution:

Enter R1: 10

Enter R2: 15

Enter R3: 20

Total resistance = 4.62 Ohms

2.8 Remarks

- Variable names are **case-sensitive** (A ≠ a).
- Always use meaningful variable names (e.g., velocity, pressure, not x1, x2).
- Use comments % to describe your code.
- Frequently save your script to prevent data loss.
- MATLAB automatically saves command history for later reuse.

2.9 Exercises

Exercise 1:

Create a script that computes the area and circumference of a circle given the radius (r).

Display both results.

Exercise 2:

Write a program to convert temperature from Celsius to Fahrenheit:

[

F = \frac{9}{5}C + 32

]

Exercise 3:

Create a complex number (Z = 3 + 4i).

Display its magnitude and phase.

Exercise 4:

Define a string variable containing your department's name and display:

Welcome to the Department of ...

2.10 Solutions

Solution 1:

```
r = input('Enter radius: ');
A = pi * r^2;
C = 2 * pi * r;
fprintf('Area = %.2f, Circumference = %.2f\n', A, C);
```

Solution 2:

```
C = input('Enter temperature in Celsius: ');
```

```
F = (9/5)*C + 32;
fprintf('Temperature in Fahrenheit = %.2f\n', F);
```

Solution 3:

```
Z = 3 + 4i;
mag = abs(Z);
phi = angle(Z);
fprintf('Magnitude = %.2f, Phase = %.2f radians\n', mag, phi);
```

Solution 4:

```
dept = 'Mechanical Engineering';
disp(['Welcome to the Department of ', dept]);
```

Chapter 3: Reading, Displaying, and Saving Data in MATLAB

3.1 Introduction

In MATLAB, data manipulation plays a fundamental role in engineering and scientific computing.

Whether working with experimental measurements, simulation results, or data files, it is essential to know how to **read**, **display**, and **save** data efficiently.

This chapter introduces the basic techniques and commands that allow users to import data from various file types, display it in a readable form, and store it for future use.

By the end of this chapter, you should be able to:

- Read data from text and binary files.
- Display data in the Command Window and graphically.
- Save and export data to files in different formats.

3.2 Reading Data

MATLAB provides several functions to import data from different file formats such as **.txt**, **.csv**, **.dat**, or **.xls**.

The choice of function depends on the type of data and the structure of the file.

3.2.1 The load Command

The load command is one of the simplest ways to import numerical data stored in a text or MAT-file.

Syntax:

load filename

load('filename')

If the file contains only numeric data arranged in columns, MATLAB automatically loads it into a variable with the same name as the file.

Example:

load data1.txt

This command loads the contents of *data1.txt* into a variable called **data1**.

If the file contains multiple variables in MATLAB format (.mat), you can load specific ones:

load('experiment.mat', 'temperature', 'pressure')

3.2.2 Reading Text Files with fopen and fscanf

For more control, MATLAB allows reading data using **low-level file I/O functions**.

Example:

```
fid = fopen('results.txt','r');     % Open file in read mode
```

```
A = fscanf(fid, '%f', [3, 4]);     % Read numeric data into a 3x4 matrix
fclose(fid);                       % Close file
```

Here:

- fopen opens the file.
- fscanf reads formatted data.
- fclose closes the file after reading.

If the file contains mixed types of data (numbers and text), use textscan instead.

3.2.3 Reading from Excel Files

MATLAB has built-in support for Excel files.

Example:

```
data = xlsread('measurements.xlsx');
```

You can also specify the sheet and range:

```
data = xlsread('measurements.xlsx', 'Sheet1', 'A1:C10');
```

3.2.4 Import Tool

MATLAB also provides a **graphical import tool**.

You can open it by typing:

```
uiimport
```

This tool lets you visually select files, preview their contents, and choose import options without writing code.

3.3 Displaying Data

Once the data is loaded, MATLAB provides various methods for displaying it in text or graphical form.

3.3.1 Command Window Display

To display variables directly in the Command Window:

```
disp(A)
```

Alternatively, you can print formatted text using:

```
fprintf('The average value is %.2f\n', mean(A));
```

3.3.2 Graphical Display

Data visualization is a powerful feature of MATLAB.

You can create simple 2D and 3D plots easily.

Examples:

```
x = 0:0.1:10;
y = sin(x);
plot(x, y)
```

title('Sine Function')

xlabel('x')

ylabel('sin(x)')

grid on

For multiple data sets:

plot(x, sin(x), 'r', x, cos(x), 'b--')

legend('sin(x)', 'cos(x)')

3.4 Saving Data

After processing or analysis, it is often necessary to save results for future use.

3.4.1 The save Command

The simplest way to save data is using save.

Syntax:

save filename

save('filename')

This command saves all workspace variables to a binary MAT-file.

Example:

save results

This creates a file named *results.mat* containing all variables.

To save specific variables:

save results A B

To save in ASCII format (human-readable):

save results.txt A -ascii

3.4.2 Writing to Text Files with fprintf

You can also export formatted data to text files.

Example:

fid = fopen('output.txt','w');

fprintf(fid, 'Temperature = %.2f °C\n', 25.78);

fclose(fid);

For matrices:

fid = fopen('matrix.txt','w');

fprintf(fid,'%6.2f %6.2f %6.2f\n', A');

fclose(fid);

3.4.3 Writing to Excel Files

Use the xlswrite function to export data to Excel:

xlswrite('results.xlsx', A)

You can specify the sheet and range:

xlswrite('results.xlsx', A, 'Sheet1', 'B2')

3.5 Remarks

- Always close files after reading or writing with fclose(fid).
- Avoid overwriting important data files — verify the filename before saving.
- For large datasets, MAT-files are faster and more memory-efficient than text files.
- Use comments (%) in your scripts to explain each step.

3.6 Practical Example

Example: Reading, Processing, and Saving Data

```
% Step 1: Read data from a text file
data = load('temperature.txt');
% Step 2: Display basic information
disp('Temperature data loaded successfully:');
disp(data);
% Step 3: Compute statistics
avg = mean(data);
maxT = max(data);
minT = min(data);
fprintf('Average Temperature: %.2f°C\n', avg);
fprintf('Maximum Temperature: %.2f°C\n', maxT);
fprintf('Minimum Temperature: %.2f°C\n', minT);
% Step 4: Plot results
plot(data, 'r-o')
title('Temperature Variation')
xlabel('Measurement Number')
ylabel('Temperature (°C)')
grid on
% Step 5: Save results
save('temperature_results.mat', 'avg', 'maxT', 'minT');
```

3.7 Exercises

Exercise 1:

Create a text file named *data.txt* containing 3 columns and 10 rows of numeric values.

Write a MATLAB script to:

1. Read the file.
2. Compute the mean of each column.
3. Display the results using fprintf.
4. Save the means in a file named *mean_values.txt*.

Exercise 2:

Import data from an Excel file named *measurements.xlsx*, sheet “Test1”.

Plot the first column versus the second column and label the axes appropriately.

Save the figure as *plot1.png*.

Exercise 3:

Write a MATLAB script that:

1. Generates random numbers using rand(1,10).
2. Displays them in the Command Window.
3. Saves them to a MAT-file and also to a text file.
4. Reads them again and verifies the values.

Solutions (Suggested)

Exercise 1 Solution:

```
data = load('data.txt');
means = mean(data);
fprintf('Column Means: %.2f %.2f %.2f\n', means);
save('mean_values.txt', 'means', '-ascii');
```

Exercise 2 Solution:

```
data = xlsread('measurements.xlsx','Test1');
plot(data(:,1), data(:,2));
xlabel('Input Variable');
ylabel('Output Variable');
title('Measurement Plot');
saveas(gcf,'plot1.png');
```

Exercise 3 Solution:

```
A = rand(1,10);
disp(A);
save('random_data.mat', 'A');
save('random_data.txt', 'A', '-ascii');
load('random_data.mat');
disp(A);
```

Chapter 4: Vectors and Matrices in MATLAB

4.1 Introduction

In MATLAB, **vectors** and **matrices** are the fundamental data structures.

MATLAB itself stands for **MAT**rix **LAB**oratory, highlighting its design around matrix computations.

Every variable in MATLAB is a matrix by default—even a single number is treated as a 1×1 matrix.

Understanding how to create, manipulate, and perform operations on vectors and matrices is essential for any MATLAB user, especially in engineering and scientific applications.

This chapter provides a comprehensive overview of how to:

- Create and define vectors and matrices.
- Access and modify their elements.
- Perform arithmetic and logical operations.
- Use built-in MATLAB functions for matrix manipulation.
- Apply vector and matrix operations to solve real-world problems.

4.2 Vectors

A **vector** is a one-dimensional array of numbers.

There are two types of vectors in MATLAB:

- **Row vectors** (1×n)
- **Column vectors** (n×1)

4.2.1 Creating Vectors

(a) Using square brackets

To create a vector, use square brackets [ ] and separate elements by spaces or commas.

Example:

```
v = [1 2 3 4 5];
```

This creates a **row vector** with 5 elements.

To create a **column vector**, use semicolons ; between elements:

```
v = [1; 2; 3; 4; 5];
```

(b) Using the colon operator

The colon operator (:) is a convenient way to generate regularly spaced vectors.

Syntax:

```
start:step:end
```

Examples:

```
v = 1:5          % [1 2 3 4 5]
v = 0:2:10       % [0 2 4 6 8 10]
```

v = 10:-1:5 % [10 9 8 7 6 5]

(c) Using built-in functions

MATLAB provides several functions to create specific types of vectors:

Function	Description	Example
linspace(a,b,n)	Generates n equally spaced points between a and b	linspace(0,1,5) → [0 0.25 0.5 0.75 1]
logspace(a,b,n)	Generates logarithmically spaced points between 10^a and 10^b	logspace(1,3,3) → [10 100 1000]

4.2.2 Accessing Elements

Use parentheses ( ) with indices to access or modify vector elements.

Example:

v = [3 6 9 12];

v(2) % returns 6

v(4) = 15; % changes the 4th element to 15

You can also access multiple elements:

v([1 3]) % returns elements 1 and 3 → [3 9]

4.2.3 Vector Operations

MATLAB supports **element-by-element** and **vectorized** operations.

Example:

a = [1 2 3];

b = [4 5 6];

c = a + b; % [5 7 9]

d = a .* b; % element-wise multiplication → [4 10 18]

e = a .^ 2; % element-wise square → [1 4 9]

4.3 Matrices

A **matrix** is a two-dimensional array of numbers arranged in rows and columns.

4.3.1 Creating Matrices

Use square brackets [ ] and separate columns with spaces or commas, and rows with semicolons.

Example:

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

This creates a **3×3 matrix**.

You can also create special matrices using MATLAB functions:

Function	Description	Example
zeros(m,n)	m×n matrix of zeros	zeros(3,2) → 3×2 matrix of zeros
ones(m,n)	m×n matrix of ones	ones(2,4)
eye(n)	n×n identity matrix	eye(3)
rand(m,n)	random matrix (uniform distribution)	rand(2,3)
diag(v)	creates diagonal matrix from vector v	diag([1 2 3])

4.3.2 Accessing Matrix Elements

Matrix elements are accessed using **row, column** indices:

Example:

```
A = [10 20 30; 40 50 60; 70 80 90];
A(2,3)      % element in row 2, column 3 → 60
A(1,:)      % first row → [10 20 30]
A(:,2)      % second column → [20; 50; 80]
```

You can also extract submatrices:

```
B = A(1:2, 2:3)   % top-right 2×2 block
```

4.3.3 Matrix Operations

MATLAB is optimized for **matrix algebra**.

The following operations are commonly used:

Operation	Symbol	Example
Addition/Subtraction	+, -	A + B
Matrix Multiplication	*	A * B
Element-wise Multiplication	.*	A .* B
Power	^ or .^	A^2, A.^2
Transpose	'	A'
Inverse	inv(A)	
Determinant	det(A)	

Example:

```
A = [1 2; 3 4];
B = [5 6; 7 8];
C = A * B;          % matrix multiplication
```

```
D = A .* B;        % element-wise multiplication
T = A';            % transpose
I = inv(A);        % inverse
```

4.4 Matrix Functions

MATLAB includes a rich set of built-in matrix manipulation functions:

Function	Description
size(A)	returns matrix dimensions
length(A)	returns the longest dimension
numel(A)	total number of elements
sum(A)	sum of each column
mean(A)	mean of each column
max(A)	maximum element in each column
min(A)	minimum element
reshape(A,m,n)	reshapes A into m×n matrix
sort(A)	sorts each column
cat(dim,A,B)	concatenates matrices along dimension dim

Example:

```
A = [1 2; 3 4];
B = [5 6; 7 8];
C = cat(1,A,B);   % vertical concatenation
D = cat(2,A,B);   % horizontal concatenation
```

4.5 Special Matrices and Functions

Matrix	Description
magic(n)	creates a magic square (rows and columns sum to same value)
hilb(n)	Hilbert matrix (used in numerical analysis)
randn(m,n)	random matrix (normal distribution)
eye(n)	identity matrix

Example:

```
M = magic(3)
H = hilb(4)
```

4.6 Remarks

- Always ensure matrices have compatible dimensions before performing operations.
- MATLAB performs operations **column-wise** by default.
- To apply a function to every element, use the dot operator (e.g., .^, .*).
- Be careful when transposing complex matrices:
    - ' performs **complex conjugate transpose**
    - .' performs **non-conjugate transpose**

4.7 Practical Example

```
% Create two matrices
A = [1 2 3; 4 5 6; 7 8 9];
B = [9 8 7; 6 5 4; 3 2 1];

% Compute operations
C1 = A + B;          % Addition
C2 = A .* B;         % Element-wise product
C3 = A * B;          % Matrix multiplication
C4 = A';             % Transpose
detA = det(A);       % Determinant

% Display results
disp('Matrix A + B ='); disp(C1);
disp('Element-wise product A.*B ='); disp(C2);
disp('Matrix product A*B ='); disp(C3);
disp('Transpose of A ='); disp(C4);
fprintf('Determinant of A = %.2f\n', detA);
```

4.8 Exercises

Exercise 1:

Create a row vector x from 1 to 20 with a step of 2.

Create a column vector y with elements from 5 to 15.

Find:

1. The element-wise sum x + y'
2. The product x .* y'

Exercise 2:

Given a 3×3 matrix

[

A = \begin{bmatrix} 2 & 4 & 6 \ 1 & 3 & 5 \ 7 & 8 & 9 \end{bmatrix}

]

Compute:

- The transpose of A
- The determinant of A
- The inverse of A
- The sum of all elements

Exercise 3:

Use MATLAB to:

1. Create a random 4×4 matrix using rand.
2. Compute its mean value using mean.
3. Extract the 2nd row and 3rd column.
4. Replace all elements greater than 0.5 with 1.

Solutions (Suggested)

Exercise 1:

```
x = 1:2:20;
y = (5:15)';
S = x + y';
P = x .* y';
```

Exercise 2:

```
A = [2 4 6; 1 3 5; 7 8 9];
At = A';
d = det(A);
invA = inv(A);
s = sum(A(:));
```

Exercise 3:

```
M = rand(4,4);
avg = mean(M(:));
r2 = M(2,:);
c3 = M(:,3);
M(M > 0.5) = 1;
```

Chapter 5: Control Instructions in MATLAB

5.1 Introduction

In programming, **control instructions** (or control structures) are essential for determining the flow of execution in a program.

They allow MATLAB to **make decisions**, **repeat operations**, and **handle errors** based on conditions or logic.

In MATLAB, control structures include:

- **Conditional statements**: if, else, elseif, switch, case, otherwise
- **Loop structures**: for, while
- **Control keywords**: break, continue, and return
- **Error handling**: try, catch

This chapter explains how to use these instructions effectively with detailed examples and practical exercises.

5.2 Conditional Statements

Conditional statements enable MATLAB to **execute specific code blocks** based on whether a condition is true or false.

5.2.1 The if ... end Statement

Syntax:

```
if condition
   statements
end
```

Example:

```
x = 10;
if x > 0
   disp('x is positive');
end
```

5.2.2 The if ... else Statement

If the condition is not satisfied, MATLAB executes the code in the else block.

Example:

```
x = -3;
if x >= 0
   disp('x is positive or zero');
else
   disp('x is negative');
end
```

5.2.3 The if ... elseif ... else Statement

Use elseif when multiple conditions need to be tested sequentially.

Example:

```
x = 0;
if x > 0
   disp('Positive number');
elseif x < 0
   disp('Negative number');
else
   disp('Zero');
end
```

5.2.4 Logical Operators

Operator	Meaning	Example	Result
==	Equal to	x == y	true if x equals y
~=	Not equal	x ~= y	true if x ≠ y
<	Less than	x < y	true if x < y
>	Greater than	x > y	true if x > y
<=	Less or equal	x <= y	true if x ≤ y
>=	Greater or equal	x >= y	true if x ≥ y
&&	Logical AND	(x>0)&&(y>0)	true if both true
`		`	Logical OR
~	Logical NOT	~(x>0)	negates condition

5.3 The switch Statement

The switch structure is used when a variable or expression can take **multiple discrete values**.

It is an alternative to writing many if...elseif conditions.

Syntax:

```
switch variable
   case value1
      statements
   case value2
      statements
   otherwise
```

```
    statements
end
```

Example:

```
grade = 'B';

switch grade
   case 'A'
      disp('Excellent');
   case 'B'
      disp('Good');
   case 'C'
      disp('Average');
   otherwise
      disp('Invalid grade');
end
```

5.4 Loop Structures

Loops allow MATLAB to **repeat a set of instructions** several times, either for a known number of iterations or until a condition is met.

5.4.1 The for Loop

Used when the number of iterations is known in advance.

Syntax:

```
for variable = start:step:end
   statements
end
```

Example 1:

```
for k = 1:5
   fprintf('Iteration %d\n', k);
end
```

Example 2:

```
sum = 0;
for i = 1:10
   sum = sum + i;
end
disp(sum);
```

5.4.2 The while Loop

Used when the number of iterations is **not known in advance**, and repetition continues **while a condition is true**.

Syntax:

```
while condition
   statements
end
```

Example:

```
x = 1;
while x < 10
   disp(x);
   x = x + 2;
end
```

5.4.3 Nested Loops

Loops can be placed **inside one another**, creating **nested loops**.

Example:

```
for i = 1:3
   for j = 1:2
      fprintf('i = %d, j = %d\n', i, j);
   end
end
```

5.5 Loop Control Keywords

MATLAB provides commands to alter the flow inside loops.

Command	Description
break	Terminates the loop immediately
continue	Skips the remaining code in the loop and proceeds to the next iteration
return	Exits from a function or script

Example using break:

```
for i = 1:10
   if i == 5
      break;
   end
   disp(i);
```

```
end
```

Example using continue:

```
for i = 1:5
   if i == 3
      continue;
   end
   disp(i);
end
```

5.6 Error Handling with try ... catch

When an error occurs during execution, MATLAB stops the program.

To prevent abrupt termination, use the **try...catch** structure.

Syntax:

```
try
   statements
catch exception
   statements
end
```

Example:

```
try
   A = [1 2; 3 4];
   B = [1 2 3];
   C = A * B;   % This will cause an error
catch
   disp('Matrix dimensions do not match!');
end
```

5.7 Combining Control Structures

You can combine if, for, and while statements for more complex logic.

Example:

```
for i = 1:5
   if mod(i,2)==0
      disp(['Even number: ', num2str(i)]);
   else
      disp(['Odd number: ', num2str(i)]);
   end
```

```
end
```

5.8 Practical Examples

Example 1: Factorial Calculation

```
n = 5;
fact = 1;
for i = 1:n
    fact = fact * i;
end
fprintf('Factorial of %d is %d\n', n, fact);
```

Example 2: Average of Positive Numbers

```
numbers = [-2 5 8 -1 4 0];
sumPos = 0; count = 0;

for i = 1:length(numbers)
    if numbers(i) > 0
        sumPos = sumPos + numbers(i);
        count = count + 1;
    end
end

if count > 0
    avg = sumPos / count;
    fprintf('Average of positive numbers = %.2f\n', avg);
else
    disp('No positive numbers found');
end
```

Example 3: While Loop with User Input

```
x = input('Enter a number (0 to stop): ');
sum = 0;

while x ~= 0
    sum = sum + x;
    x = input('Enter a number (0 to stop): ');
end
```

```
fprintf('The total sum is %.2f\n', sum);
```

5.9 Remarks

- Always ensure that loops have **termination conditions**, otherwise infinite loops occur.
- for loops are preferable when the number of iterations is **known**.
- while loops are suitable when repetition depends on a **condition**.
- The use of try...catch is recommended for programs handling user input or file operations.
- Proper indentation makes control structures more readable and reduces errors.

5.10 Exercises

Exercise 1:

Write a MATLAB program that asks the user to enter a number and displays whether it is **positive**, **negative**, or **zero** using an if statement.

Exercise 2:

Write a program that calculates the **sum of all even numbers** between 1 and 50 using a for loop.

Exercise 3:

Create a program that uses a while loop to count how many numbers a user enters before typing -1.

Exercise 4:

Use a switch statement to print the name of the day of the week given a number between 1 and 7.

Exercise 5:

Write a MATLAB script that attempts to divide two numbers and handles division-by-zero errors using try...catch.

Solutions (Suggested)

Exercise 1:

```
x = input('Enter a number: ');
if x > 0
   disp('Positive');
elseif x < 0
   disp('Negative');
else
```

```
    disp('Zero');
end
```

Exercise 2:

```
sum = 0;
for i = 1:50
    if mod(i,2) == 0
        sum = sum + i;
    end
end
disp(sum);
```

Exercise 3:

```
count = 0;
num = input('Enter a number (-1 to stop): ');
while num ~= -1
    count = count + 1;
    num = input('Enter a number (-1 to stop): ');
end
fprintf('You entered %d numbers.\n', count);
```

Exercise 4:

```
day = input('Enter a number (1-7): ');
switch day
    case 1
        disp('Monday');
    case 2
        disp('Tuesday');
    case 3
        disp('Wednesday');
    case 4
        disp('Thursday');
    case 5
        disp('Friday');
    case 6
        disp('Saturday');
    case 7
```

```
        disp('Sunday');
    otherwise
        disp('Invalid input');
end**Exercise 5:**
try
    a = input('Enter numerator: ');
    b = input('Enter denominator: ');
    result = a / b;
    fprintf('Result = %.2f\n', result);
catch
    disp('Error: Division by zero!');
end
```

Chapter 6: Function Files in MATLAB

6.1 Introduction

In MATLAB, a **function** is a group of statements that together perform a specific task.

Functions help to:

- Make programs **modular** (divided into logical parts)
- **Reuse code** easily
- **Simplify debugging**
- Improve **readability** and **organization**

Unlike scripts, which run commands directly in the MATLAB workspace, **function files** have their own **workspace** and only share data through **input and output arguments**.

6.2 Script vs Function

Feature	Script	Function
Workspace	Uses the **base workspace**	Has its **own local workspace**
Inputs	None (uses existing variables)	Accepts **input arguments**
Outputs	None (creates variables in base workspace)	Returns **output arguments**
Syntax	Sequence of commands	Starts with the function keyword
File name	Any name	Must match the function name

6.3 Creating a Function File

A function file is a **MATLAB file** (.m extension) that starts with the keyword **function**.

General Syntax:

```
function [output1, output2, ...] = function_name(input1, input2, ...)
  % Function body (instructions)
end
```

Rules:

1. The file must be saved with the same name as the function.
   → For example, if the function is called mySum, save it as **mySum.m**
2. Comments at the beginning of the file serve as **help text**.
   → They can be accessed with the MATLAB command:
3. help function_name

6.4 Example: A Simple Function

Example 1: Sum of Two Numbers

```
function s = mySum(a, b)
% mySum calculates the sum of two numbers
s = a + b;
```

```
end
```

Execution:

```
>> result = mySum(4, 7)
result =
   11
```

6.5 Function with Multiple Outputs

Functions can return more than one value.

Example 2:

```
function [sum, diff, prod] = operations(a, b)
% operations performs basic arithmetic operations
sum = a + b;
diff = a - b;
prod = a * b;
end
```

Execution:

```
>> [s, d, p] = operations(5, 3)
s = 8
d = 2
p = 15
```

6.6 Local and Global Variables

Each function has its own **local workspace**.

Variables defined inside a function are **not accessible** outside it.

Global Variables

To share a variable between multiple functions, use the keyword global.

Example:

```
global x
x = 10;
myFunction
In the function file myFunction.m:
function myFunction
global x
disp(['The global variable x = ', num2str(x)]);
end
```

6.7 Nested and Anonymous Functions

6.7.1 Nested Functions

A **nested function** is defined inside another function.

It can access variables of the parent function.

Example:

```
function outerFunction
x = 5;
nestedFunction
   function nestedFunction
      disp(['x inside nested function = ', num2str(x)]);
   end
end
```

6.7.2 Anonymous Functions

These are **short, one-line functions** defined without a separate file.

Syntax:

```
f = @(x) expression
```

Example:

```
f = @(x) x.^2 + 3*x + 2;
y = f(4)
```

Result:

```
y = 30
```

6.8 Built-in Functions

MATLAB includes many predefined (built-in) functions such as:

- sin(x), cos(x), sqrt(x), log(x)
- mean(x), sum(x), max(x), min(x)
- size(A), length(A)

These are ready-to-use and optimized for performance.

However, users can define **custom functions** for specific needs.

6.9 Input/Output Arguments

A function can handle **any number of inputs or outputs**.

Example:

```
function y = average(varargin)
% Computes the average of any number of inputs
n = length(varargin);
```

```
sum = 0;
for k = 1:n
   sum = sum + varargin{k};
end
y = sum / n;
end
```

Execution:

```
>> avg = average(2, 4, 6, 8)
avg =
    5
```

6.10 Function Documentation

It is good practice to include a **header comment** that explains:

- The purpose of the function
- The meaning of inputs and outputs
- Example of usage

Example:

```
function area = circleArea(r)
% circleArea computes the area of a circle
% Input: r – radius
% Output: area – computed area
% Example: circleArea(3)
area = pi * r^2;
end
```

To view documentation:

```
>> help circleArea
```

6.11 Subfunctions

Multiple functions can be placed in the same file.

Only the **first** function (the main one) is visible from outside the file;

others are called **subfunctions**.

Example:

```
function result = mainFunction(x)
result = square(x) + cube(x);

function y = square(a)
```

```
    y = a^2;
end


function z = cube(a)
    z = a^3;
end
end
```

Execution:

```
>> mainFunction(2)
ans = 12
```

6.12 Error Checking in Functions

It is often useful to check for invalid input values using nargin, nargout, or error().

Example:

```
function y = divide(a, b)
% Checks for division by zero
if nargin < 2
    error('Two inputs are required');
elseif b == 0
    error('Division by zero is not allowed');
else
    y = a / b;
end
end
```

6.13 Example: Quadratic Equation Solver

File: quadSolver.m

```
function [x1, x2] = quadSolver(a, b, c)
% Solves ax^2 + bx + c = 0
% Returns two roots (real or complex)
delta = b^2 - 4*a*c;
if delta >= 0
    x1 = (-b + sqrt(delta)) / (2*a);
    x2 = (-b - sqrt(delta)) / (2*a);
else
    x1 = (-b + sqrt(delta)) / (2*a);
```

```
    x2 = (-b - sqrt(delta)) / (2*a);
    disp('Complex roots');
end
end
```

Execution:

```
>> [x1, x2] = quadSolver(1, -3, 2)
x1 = 2
x2 = 1
```

6.14 Practical Exercises

Exercise 1:

Create a function circle.m that takes the radius as input and returns the **area** and **circumference**.

Exercise 2:

Write a function fahrenheitToCelsius.m that converts temperature from Fahrenheit to Celsius.

Exercise 3:

Create a function isPrime.m that determines whether a number is prime.

Exercise 4:

Write a function factorialRec.m that computes the factorial of a number **recursively**.

Exercise 5:

Design a function matrixStats.m that takes a matrix and returns its **mean**, **max**, and **min** values.

6.15 Suggested Solutions

1. circle.m

```
function [area, circ] = circle(r)
area = pi * r^2;
circ = 2 * pi * r;
end
```

2. fahrenheitToCelsius.m

```
function C = fahrenheitToCelsius(F)
C = (F - 32) * 5/9;
end
```

3. isPrime.m

```
function result = isPrime(n)
if n < 2
```

```
    result = false;
    return;
end
for i = 2:sqrt(n)
    if mod(n,i) == 0
        result = false;
        return;
    end
end
result = true;
end
```

4. factorialRec.m

```
function f = factorialRec(n)
if n == 0
    f = 1;
else
    f = n * factorialRec(n-1);
end
end
```

5. matrixStats.m

```
function [m, M, mmin] = matrixStats(A)
m = mean(A(:));
M = max(A(:));
mmin = min(A(:));
end
```

Chapter 7: Graphics in MATLAB

7.1 Introduction

One of MATLAB's most powerful features is its ability to produce **high-quality graphics** for data visualization and scientific analysis.

Graphics in MATLAB allow users to:

- Plot data and mathematical functions
- Visualize matrices and 3D surfaces
- Annotate and customize figures
- Export graphs for reports and publications

MATLAB provides **two main types** of graphics:

1. **2D graphics** (line plots, bar charts, histograms, etc.)
2. **3D graphics** (surfaces, meshes, contours, etc.)

7.2 Basic 2D Plots

7.2.1 The plot Function

The most commonly used function for 2D plotting is plot(x, y).

Example:

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x, y);
```

This command plots the sine function between 0 and (2\pi).

7.2.2 Adding Titles, Labels, and Grid

To make plots more informative, we can add:

- A title using title('text')
- Axis labels using xlabel() and ylabel()
- A grid using grid on

Example:

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x, y, 'r', 'LineWidth', 2);
title('Sine Function');
xlabel('x (radians)');
ylabel('sin(x)');
grid on;
```

7.2.3 Multiple Curves on the Same Plot

You can plot several curves in the same figure using hold on.

Example:

```
x = 0:0.1:2*pi;
plot(x, sin(x), 'r', 'LineWidth', 2);
hold on;
plot(x, cos(x), 'b--', 'LineWidth', 2);
legend('sin(x)', 'cos(x)');
title('Sine and Cosine Functions');
grid on;
hold off;
```

7.2.4 Plot Customization

Parameter	Description	Example
'r', 'b', 'g'	Red, Blue, Green	'r'
'--', ':', '-.'	Line style	'--'
'o', '*', 's'	Marker type	'o'
'LineWidth'	Thickness	'LineWidth', 2

Example:

```
x = 0:0.2:4*pi;
y = sin(x);
plot(x, y, 'g--o', 'LineWidth', 1.5);
```

7.3 Subplots

To display multiple plots in one figure, use the command subplot(m, n, p)

where:

- m = number of rows
- n = number of columns
- p = position index

Example:

```
x = 0:0.1:2*pi;
subplot(2,1,1);
plot(x, sin(x));
title('Sine Function');

subplot(2,1,2);
plot(x, cos(x));
```

```
title('Cosine Function');
```

7.4 Specialized 2D Plots

7.4.1 Bar Charts

```
x = [1 2 3 4];
y = [5 8 2 9];
bar(x, y);
title('Bar Chart Example');
```

7.4.2 Histogram

```
data = randn(1, 1000);
histogram(data, 20);
title('Histogram of Random Data');
```

7.4.3 Pie Charts

```
values = [30 10 20 40];
labels = {'A','B','C','D'};
pie(values, labels);
title('Pie Chart Example');
```

7.5 3D Graphics

MATLAB provides functions to visualize 3D data, surfaces, and mathematical functions.

7.5.1 3D Line Plot

```
t = 0:0.1:10;
plot3(t, sin(t), cos(t));
xlabel('t'); ylabel('sin(t)'); zlabel('cos(t)');
title('3D Helix Curve');
grid on;
```

7.5.2 Mesh and Surface Plots

To visualize functions of two variables (z = f(x, y)), use mesh or surf.

Example:

```
[x, y] = meshgrid(-3:0.1:3, -3:0.1:3);
z = sin(sqrt(x.^2 + y.^2));
surf(x, y, z);
title('3D Surface Plot');
xlabel('x'); ylabel('y'); zlabel('z');
```

Mesh Example:

```
mesh(x, y, z);
```

```
title('3D Mesh Plot');
```

7.5.3 Contour Plots

Contour plots display **level curves** of a 3D surface on a 2D plane.

```
[x, y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x.^2 + y.^2;
contour(x, y, z, 20);
title('Contour Plot');
xlabel('x'); ylabel('y');
```

7.6 Image and Color Control

7.6.1 Using colormap

The color scheme can be changed using colormap.

Example:

```
surf(peaks);
colormap jet;
colorbar;
```

Common colormaps:

- jet, parula, hsv, gray, hot, cool

7.6.2 Colorbars and Shading

```
surf(peaks);
shading interp;
colorbar;
title('Smooth Surface with Colorbar');
```

7.7 Figure Management

MATLAB allows you to create and manage multiple figures simultaneously.

Example:

```
figure(1);
plot(sin(0:0.1:2*pi));
figure(2);
plot(cos(0:0.1:2*pi));
```

Use:

- close all → Close all figures
- clf → Clear the current figure
- saveas(gcf,'figure.png') → Save the figure as an image

7.8 Plot Annotations

You can add **text, arrows, and boxes** to annotate plots.

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x, y);
title('Annotated Sine Function');
xlabel('x'); ylabel('sin(x)');
text(pi/2, 1, 'Maximum', 'FontSize', 12, 'Color', 'red');
```

7.9 Animated Plots

Animations can be created by updating the plot in a loop.

```
x = 0:0.1:2*pi;
for k = 1:length(x)
    plot(x(1:k), sin(x(1:k)), 'b', 'LineWidth', 2);
    axis([0 2*pi -1 1]);
    pause(0.05);
end
```

7.10 Exporting Figures

MATLAB supports exporting plots in several formats:

```
saveas(gcf, 'plot1.png');     % Save as PNG
saveas(gcf, 'plot1.fig');     % Save as MATLAB figure
print('plot1','-dpdf');       % Save as PDF
```

7.11 Practical Examples

Example 1 – Damped Oscillation

```
t = 0:0.1:10;
y = exp(-0.1*t).*sin(2*t);
plot(t, y);
title('Damped Oscillation');
xlabel('Time'); ylabel('Amplitude');
grid on;
```

Example 2 – 3D Surface with Lighting

```
[x, y] = meshgrid(-3:0.1:3);
z = sin(x).*cos(y);
surf(x, y, z);
shading interp;
```

```
lighting phong;
title('3D Surface with Lighting');
```

7.12 Remarks

- MATLAB graphics are **vector-based**, producing high-quality output.
- Always label axes and include a legend or title.
- Use hold on to plot multiple datasets on the same graph.
- For large data visualization, use interactive tools like **plottools** or **figure menu**.

7.13 Exercises

Exercise 1:

Plot the functions (y = x^2) and (y = x^3) on the same figure with different colors and a legend.

Exercise 2:

Create a 3D surface for (z = e^{-x^2 - y^2}) using mesh and surf.

Exercise 3:

Generate a bar chart showing the population of 5 cities.

Exercise 4:

Plot a sine wave and annotate the maximum and minimum points.

Exercise 5:

Write a script that animates the rotation of a 3D surface.

7.14 Suggested Solutions

Exercise 1:

```
x = -2:0.1:2;
plot(x, x.^2, 'r', x, x.^3, 'b--');
legend('x^2', 'x^3');
title('Comparison of x^2 and x^3');
```

Exercise 2:

```
[x, y] = meshgrid(-2:0.1:2);
z = exp(-x.^2 - y.^2);
surf(x, y, z);
title('3D Gaussian Surface');
```

Exercise 3:

```
cities = {'Paris','London','Rome','Berlin','Madrid'};
population = [11 9 4 6 7];
bar(population);
```

```
set(gca, 'XTickLabel', cities);
title('City Populations');
```

Exercise 4:

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x, y);
hold on;
[~, imax] = max(y);
[~, imin] = min(y);
plot(x(imax), y(imax), 'ro', x(imin), y(imin), 'bo');
text(x(imax), y(imax), ' Max');
text(x(imin), y(imin), ' Min');
title('Sine Wave with Annotations');
```

Exercise 5:

```
[x, y] = meshgrid(-2:0.1:2);
z = sin(x).*cos(y);
for angle = 0:5:360
    surf(x, y, z);
    view(angle, 30);
    pause(0.1);
end
```

Chapter 8: Using MATLAB Toolboxes

8.1 Introduction

MATLAB is not only a powerful numerical computing environment but also a flexible platform that can be extended through **toolboxes**. A toolbox in MATLAB is a collection of specialized functions (M-files) designed to perform specific tasks in various engineering and scientific fields. Each toolbox focuses on a particular area such as signal processing, control systems, optimization, image analysis, or machine learning.

Toolboxes are extremely useful for engineers because they save development time and provide reliable, pre-tested algorithms that can be easily integrated into MATLAB scripts, functions, or Simulink models.

In mechanical engineering, toolboxes are widely used for vibration analysis, dynamic modeling, system control, optimization of parameters, and simulation.

8.1 Installing and Managing Toolboxes

MATLAB allows users to install and manage toolboxes through several methods:

a) Using the Add-On Explorer

1. Go to the **Home** tab on MATLAB's toolbar.
2. Click **Add-Ons → Get Add-Ons**.
3. Search for the desired toolbox (e.g., *Control System Toolbox*).
4. Click **Install** and MATLAB will automatically download and install it.

b) Using the Command Window

You can check the installed toolboxes using:

ver

To see a list of all available functions in a toolbox:

help toolboxname

Example:

help signal

If a toolbox is missing, MATLAB will show an error message such as:

Undefined function or variable 'tf'.

which means that the *Control System Toolbox* is not installed.

8.2 Common MATLAB Toolboxes

Below are the most frequently used toolboxes in mechanical and scientific applications.

8.2.1 Signal Processing Toolbox

This toolbox provides tools for analyzing, filtering, and visualizing signals.

Key Functions:

```
fft(x)      % Fast Fourier Transform
```

```
ifft(X)       % Inverse FFT
filter(b,a,x) % Applies a digital filter
spectrogram(x)% Computes the signal spectrum
```

Example:

```
t = 0:0.001:1;
x = sin(2*pi*50*t) + sin(2*pi*120*t);
Y = fft(x);
plot(abs(Y));
title('Signal Spectrum');
```

Application in Mechanics:

Used to analyze vibration data from accelerometers or rotating machinery to detect unbalance or bearing faults.

8.2.2 Image Processing Toolbox

This toolbox provides algorithms for image analysis, enhancement, and visualization.

Key Functions:

```
imread('image.jpg')
imshow(I)
imadjust(I)
edge(I, 'canny')
```

Example:

```
I = imread('metal_surface.jpg');
BW = edge(rgb2gray(I),'canny');
imshow(BW);
title('Detected Edges of Metal Surface');
```

Application in Mechanics:

Used for defect detection on surfaces, material inspection, or automated visual measurements.

8.2.3 Control System Toolbox

Essential for modeling, analyzing, and designing control systems.

Key Functions:

```
tf(num,den)      % Transfer function
step(sys)        % Step response
bode(sys)        % Frequency response
feedback(sys1,sys2)
```

Example:

```
num = [1];
den = [1 3 2];
sys = tf(num, den);
step(sys);
title('Step Response of Control System');
```

Application in Mechanics:

Used for the design and tuning of PID controllers in mechatronic systems.

8.2.4 Optimization Toolbox

Used to minimize or maximize objective functions under constraints.

Key Functions:

```
fminbnd(fun,a,b)
fminsearch(fun,x0)
fmincon(fun,x0,A,b)
```

Example:

```
fun = @(x) (x-3).^2 + 2;
x = fminbnd(fun, -5, 5)
```

Application in Mechanics:

Optimization of design parameters, such as minimizing stress or energy consumption.

8.2.5 Statistics and Machine Learning Toolbox

Provides tools for data analysis, regression, and predictive modeling.

Key Functions:

```
mean(x)
std(x)
fitlm(X,Y)
kmeans(X,k)
```

Example:

```
load fisheriris
X = meas(:,1:2);
[idx,C] = kmeans(X,3);
gscatter(X(:,1),X(:,2),idx);
```

Application in Mechanics:

Used for fault classification, predictive maintenance, and data-driven decision-making.

8.2.6 Simulink

Simulink is an interactive graphical environment for modeling, simulating, and analyzing dynamic systems.

Example:

To open Simulink:

simulink

Application in Mechanics:

Used to simulate mechanical systems, control loops, and multi-domain physical systems such as hydraulic or thermal models.

8.3 Remarks and Best Practices

- Always check toolbox licenses before use with:
- license('test','toolbox_name')
- Keep MATLAB and all toolboxes updated for maximum compatibility.
- Use which functionname to locate which toolbox a function belongs to.
- Organize your code to avoid dependency errors when sharing files.

8.4 Exercises

Exercise 1:

Use the **Signal Processing Toolbox** to analyze the vibration signal:

t = 0:0.001:1;

x = sin(2*pi*60*t) + 0.5*sin(2*pi*120*t);

Tasks:

1. Compute and plot its frequency spectrum using fft.
2. Identify the main frequency components.

Exercise 2:

Using the **Control System Toolbox**, create a transfer function:

[

H(s) = \frac{10}{s^2 + 3s + 10}

]

1. Plot the step response.
2. Determine the steady-state value.

Exercise 3:

Using the **Optimization Toolbox**, minimize the function:

[

f(x) = x^2 + 4x + 4

]

using fminbnd.

8.5 Solutions

Solution 1:

```
Y = fft(x);
f = (0:length(Y)-1);
plot(f, abs(Y));
title('Frequency Spectrum');
```

Solution 2:

```
num = [10];
den = [1 3 10];
sys = tf(num, den);
step(sys);
```

Solution 3:

```
fun = @(x) x.^2 + 4*x + 4;
xmin = fminbnd(fun, -10, 10)
```

Bibliography

1. MathWorks Inc., *MATLAB Documentation*, The MathWorks, Natick, MA, USA, 2024.
2. Biran, A. & Breiner, M., *MATLAB for Engineers*, 6th Edition, Pearson, 2020.
3. Palm, W. J., *Introduction to MATLAB for Engineers*, 5th Edition, McGraw-Hill Education, 2022.
4. Chapra, S. C., *Applied Numerical Methods with MATLAB for Engineers and Scientists*, 5th Edition, McGraw-Hill, 2023.
5. Attaway, S., *MATLAB: A Practical Introduction to Programming and Problem Solving*, 6th Edition, Butterworth-Heinemann, 2023.
6. Hanselman, D. & Littlefield, B., *Mastering MATLAB*, 8th Edition, Pearson, 2020.
7. Moore, H. H., *MATLAB for Engineers*, Prentice Hall, 2021.
8. Karris, S. T., *Introduction to Simulink with Engineering Applications*, Orchard Publications, 2019.
9. Hahn, B. D., *Essential MATLAB for Engineers and Scientists*, 8th Edition, Academic Press, 2023.
10. Moore, R., *MATLAB for Scientists and Engineers*, Prentice Hall, 2020.
11. Dym, C. L., *Principles of Mathematical Modeling*, Academic Press, 2019.
12. Rao, S. S., *Engineering Optimization: Theory and Practice*, 5th Edition, Wiley, 2019.
13. Dormand, J. R., *Numerical Methods for Differential Equations*, CRC Press, 2021.
14. MathWorks Inc., *Simulink User's Guide*, The MathWorks, Natick, MA, 2024.
15. Quarteroni, A., Sacco, R., & Saleri, F., *Numerical Mathematics*, 3rd Edition, Springer, 2020.
16. D'Errico, J., "Numerical Computing with MATLAB," *The MathWorks File Exchange*, 2023.
17. Moler, C., *Numerical Computing with MATLAB*, SIAM, 2021.
18. Chandra, R., *Parallel Programming in MATLAB*, MathWorks, 2022.
19. Gilat, A., *MATLAB: An Introduction with Applications*, 6th Edition, Wiley, 2022.
20. Pratap, R., *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers*, 6th Edition,