Department of Mechanical and Production Engineering
Ahsanullah University of Science and Technology (AUST)

IPE 3110: Programming for Engineers

Credit Hour: 1.5

Level: 3rd **Semester: 1st**

Objectives:

After completion of the course, the students will be expected to-

i. Comprehend the basic and intermediate concepts of programming.
ii. Develop the industrial engineering problem solving skills and apply them to solve a wide range of common industrial engineering problems.
iii. Analyze different problem-solving methodologies in order to implement them in different industrial engineering sectors with proficiency.

List of the Experiments:

Sl. No.	Name of the Experiment
1	Introduction to Programming Scripts, Variables and Symbolic Operators
2	Defining Functions, Surface Plotting, and Debugging Codes
3	Introduction to Matrix and Array
4	Matrix operations and manipulation
5	Auto Initialization, Indexing and Variables Manipulation
6	Importing data, reading and writing excel files and learning other miscellaneous features
7(a)	Solving Equations of Linear Algebra Using Gauss Elimination Method
7(b)	Solving Equations of Linear Algebra Using Jacobi Iteration & Gauss Seidel Iteration Method
7(c)	Solving ODEs using MATLAB
8	Methods for Solving Nonlinear Equations in MATLAB
9	Basic plotting and Curve Fitting

Books of References:

(i) Hunt B. R., Lipsman R. L., Rosenberg J M., A Guide to MATLAB, 2nd Edition, Cambridge University Press, 2006

(ii) Sayood K., Learning Programming Using MATLAB, Volume 2, 1st Edition, Morgan and Claypool Publishers, 2007

Experiment 1: Introduction to Programming Scripts, Variables and Symbolic Operators

1. Objective of the Experiment

After completing this experiment, students will be able to:

1. Understand the MATLAB working environment (desktop layout, command window, workspace, editor).
2. Create, name, save, and run MATLAB script files (.m files).
3. Use MATLAB as a calculator.
4. Declare and manipulate variables.
5. Use arithmetic, relational, and logical operators.
6. Understand vectors, matrices, and colon notation.
7. Use input/output commands (`input`, `disp`).
8. Manage workspace and understand built-in variables.

2. Introduction to the MATLAB Surface

When MATLAB starts, the default interface contains several panels:

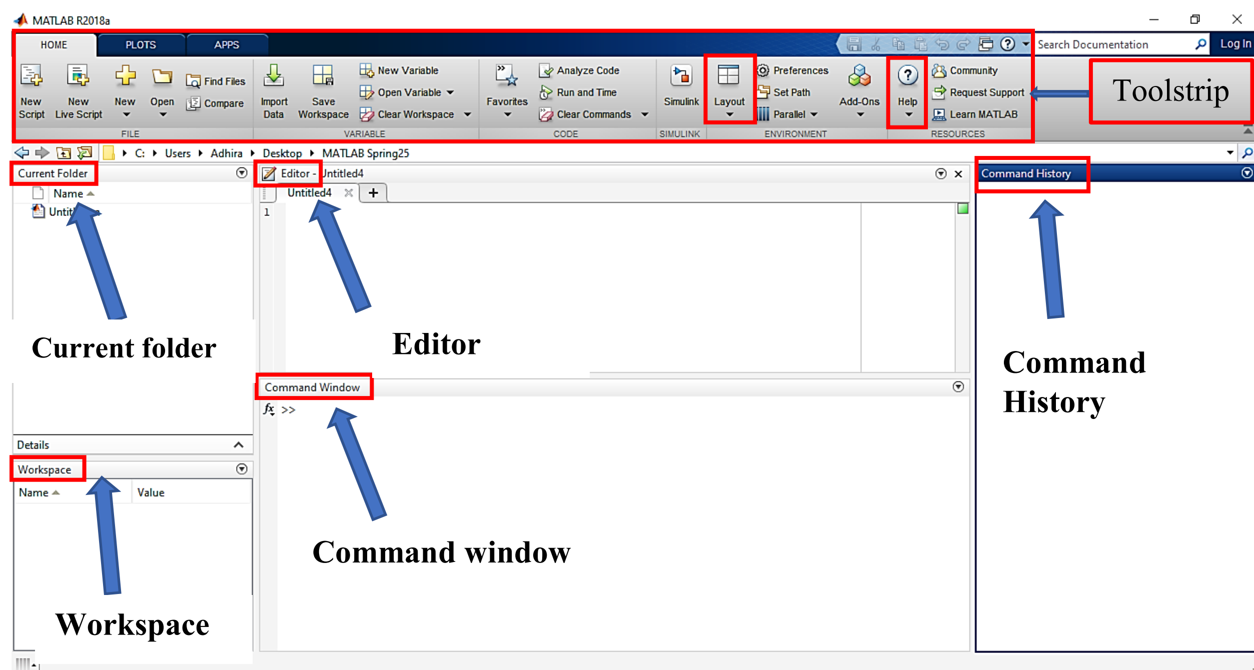


Figure 2.1: Basic Interface of MATLAB

Command Window:

- Main area where you type commands.
- Commands are executed immediately.
- MATLAB replies with output below your command.
- Prompt appears as: >>

Workspace:

- Displays all currently defined variables.
- Shows variable size, class (type), and value.
- Updates automatically as you run commands.

Current Folder / Directory:

- Shows files and folders MATLAB is currently accessing.
- MATLAB scripts must be saved in the current folder to run them.

Command History:

- Shows previously executed commands.
- You can double-click any previous command to reuse it.

Editor (Script Editor):

- If you type `edit` in command window, Editor will pop-up.
- It is used to write longer programs (scripts).
- It allows saving and running .m files.

Launch Pad / Help:

- Access documentation, examples, reference pages.
- `doc` and `help` also open these resources.

3. Creating and Saving MATLAB Files

MATLAB program files are called **scripts** and must be saved with the `.m` extension.

3.1 Creating and Saving a Script

1. Home → New Script
2. A blank editor window opens.
3. Write your MATLAB code inside it
4. Click **Save**, or press **Ctrl+S**.
5. Browse your desired folder.
6. MATLAB will ask for a filename.

3.2 Rules for Naming MATLAB Files

1. File names must start with a **letter**.
2. May contain letters, digits, and underscores only.

3. No spaces allowed.
4. File name cannot only be any number (i.e., 1.m)
5. Must not use MATLAB keywords or any in-built function (like `if`, `sin`, `plot`).
6. File name **must match the script name**.
7. File name should be meaningful.

Examples (valid):

- `experiment1.m`
- `intro_to_matlab.m`
- `myScript01.m`

Invalid:

- `1test.m` (starts with number)
- `my script.m` (contains space)
- `plot.m` (overwrites built-in function)
- `1.m` (only number)

4. All MATLAB Variables Are Matrices

In MATLAB, **all data is stored as arrays**, meaning every variable is treated as a matrix. Understanding how to create, access, and manipulate vectors and matrices is essential because they form the foundation of MATLAB programming, numerical computation, and engineering applications.

MATLAB stands for **MATrix LABoratory**, so every variable—no matter how simple—has a matrix structure.

Table: 1.1 Types of Matrices

Matrix Type	Description	Example	Notes / Usage
Scalar	A single number (1×1 matrix)	`a = 5`	Basic numeric value; MATLAB treats it as a matrix.
Row Vector	1 row, multiple columns (1×n)	`r = [1 2 3]`	Common for data sequences and indexing.
Column Vector	Multiple rows, 1 column (n×1)	`c = [1; 2; 3]`	Used in linear algebra and engineering calculations.
Square Matrix	Same number of rows and columns (n×n)	`A = [1 2; 3 4]`	Used in determinant, eigenvalue, inverse operations.
Rectangular Matrix	Rows ≠ columns (m×n)	`B = [1 2 3; 4 5 6]`	General-purpose data representation.
Zero Matrix	Matrix with all elements zero	`Z = zeros(3,3)`	Often used for initialization.
Ones Matrix	Matrix with all ones	`O = ones(2,4)`	Useful for testing operations.
Identity Matrix	Square matrix with 1s on main diagonal	`I = eye(3)`	Represents linear identity transformation.

Matrix Type	Description	Example	Notes / Usage
Diagonal Matrix	Non-zero elements only on main diagonal	`D = diag([1 2 3])`	Efficient representation; used in scaling.
Upper Triangular Matrix	All elements below main diagonal are zero	`U = triu([1 2; 3 4])`	Appears in LU factorization.
Lower Triangular Matrix	All elements above main diagonal are zero	`L = tril([1 2; 3 4])`	Used in decomposition methods.
Symmetric Matrix	$A = A^T$ (equal to its transpose)	`S = [1 2; 2 3]`	Common in engineering, statistics (covariance).
Sparse Matrix	Most elements are zero (memory-efficient)	`S = sparse([1 0; 0 0])`	Useful for large systems and graph algorithms.
Complex Matrix	Contains complex numbers	`C = [1+2i 3; 4 5-6i]`	Used in signal processing and control systems.
Random Matrix	Elements generated randomly	`R = rand(3)`	Used for simulation and testing algorithms.

5. Variables in MATLAB

A **variable** in MATLAB is a symbolic name that stores some value in the computer's memory. MATLAB uses variables extensively because *everything* in MATLAB is treated as an array (including scalars, vectors, and matrices). MATLAB automatically creates the variable once you assign a value to it. No prior declaration is needed. Understanding variables is essential because they are the foundation of all MATLAB programming.

A variable:

- Holds data (numbers, text, matrices, results of computations)
- Has a **name**, **value**, **data type**
- Has a **size and shape** (1×1, 1×n, n×1, m×n)
- Is created using the **assignment operator** =. MATLAB evaluates the expression on the right and stores the result in the variable on the left. For example:

```
x = 10;
name = "MATLAB";
A = [1 2; 3 4];
```

5.1 Rules for Variable Names

1. Begin with a–z or A–Z.
2. Remaining characters: letters, numbers, underscore.
3. MATLAB's maximum recommended length for variable name: 31 characters.
4. Remember MATLAB is case sensitive (Value ≠ value).
5. Do not reuse names of built-in variables a MATLAB keyword or function name
6. It must not contain spaces or any special characters (+, -, *, /, @, %, etc.)

Examples of valid names:

- pipeRadius
- my_value
- Temp2024

Examples of invalid names:

- 2value (cannot start with number)
- my value (contains space)
- temp-value (invalid character)
- sin (overwrites built-in function)

Table 1.2: Types of Variables

Type of Variable	Description	Example	Notes
Scalar	A single numeric value (1×1 matrix)	`x = 5;`	Simplest MATLAB variable.
Vector	A 1D array (row or column)	`v = [1 2 3];`	Row vector: 1×n, Column vector: n×1
Matrix	2D array with rows and columns	`A = [1 2; 3 4];`	MATLAB treats everything as a matrix.
String	Text enclosed in double quotes	`name = "MATLAB";`	Preferred modern text type.
Character Array	Text stored as characters in single quotes	`c = 'Hello';`	Legacy string method.
Logical Variable	Stores True/False results	`flag = (5 > 3);`	Used in conditions, loops.
Complex Variable	Number with real and imaginary parts	`z = 3 + 4i;`	MATLAB handles complex math automatically.
Row Vector / Column Vector	Structured one-dimensional arrays	`v = [1 2 3];` or `v = [1;2;3];`	Often used in engineering calculations.
Empty Variable	Variable with no elements	`x = [];`	Used to clear or initialize arrays.

Table 1.3: Variable Classes

Class Name	Description	Example	Notes
double	Default numeric data type (double precision floating point)	`x = 3.14;`	Most common type in MATLAB.
single	Single-precision numeric type	`y = single(5);`	Uses half the memory of double.
int8 / int16 / int32 / int64	Signed integer types	`a = int16(25);`	Used for integer-only data.
uint8 / uint16 / uint32 / uint64	Unsigned integer types	`b = uint8(255);`	Common in image processing.

Class Name	Description	Example	Notes
logical	Boolean true/false values	`flag = true;`	Result of comparisons.
char	Single characters or char arrays	`c = 'A';`	Used for older style strings.
string	Modern string type	`s = "Hello";`	Easier manipulation than `char`.
cell	Container for mixed-type data	`C = {1, "a", [2 3]};`	Can store different types in one array.
struct	Structure with named fields	`S.name = "Metal";`	Useful for organizing data.
table	Spreadsheet-like data container	`T = table(x,y);`	Common in data analysis.
categorical	Stores category labels	`g = categorical({'red','blue'});`	Useful for grouping data.
function_handle	Reference to a function	`f = @sin;`	Used in plotting, optimization.
datetime	Date and time information	`t = datetime;`	Used in time-series analysis.
duration	Time intervals	`d = hours(5);`	Stores time differences.
string array	Array of multiple strings	`str = ["A" "B" "C"];`	Useful for text datasets.

How to view variables:

MATLAB provides two commands to view variables:

1. who: It lists variable names.

```
>> who
Your variables are:
a b c name
```

2. whos

MATLAB shows a **table** describing:

1. Variable name
2. Size (rows × columns)
3. Number of bytes (memory usage)
4. Class (data type)
5. Attributes (if any)

It helps you understand:

- What variables exist
- Their dimensions
- How much memory they occupy
- What type of data they store

```
>> whos
Name        Size                Bytes        Class      Attributes
```

Changing (Overwriting) Variable Values:

You can assign a new value any time:

```
x = 10;
x = x + 5;
```

MATLAB updates the value immediately.

Workspace and Variable Persistence:

The **Workspace** keeps:

- All active variables
- Their values
- Their sizes
- Their types

Variables persist until:

- You clear them
- You close MATLAB
- You overwrite them

Deleting Variables: Use `clear` function to delete variables from workspace. Clearing is important to avoid accidental reuse of old values. Use `clc` to clear your command window space.

No Need for Declaration:

Unlike C/Java, MATLAB does not require:

- Data type declaration
- Memory allocation

MATLAB infers everything automatically.

5.2 Built-in Variables

MATLAB has several variables that are **predefined** and always available in the workspace. These are known as **built-in variables**. They serve special purposes and should **not be overwritten** (except a few exceptions like i and j under certain rules). Understanding these is important for early MATLAB programming, because one might often accidentally overwrite them, leading to unexpected errors.

Table 1.4: Built-in Variables

Variable	Meaning	Example Use	Warning
`ans`	Most recent answer	`5+5`	Changes automatically

pi	π constant	2*pi*r	Do not overwrite
eps	Machine precision	Accuracy checks	Don't overwrite
realmax	Largest number	Overflow tests	Read-only
realmin	Smallest number	Underflow tests	Read-only
Inf	Infinity	1/0	-
NaN	Not a number	0/0	May propagate

6. Essential MATLAB Command Window Operations

This section combines several fundamental command-line behaviors that every MATLAB beginner must know, including suppressing output, accessing documentation, and performing input/output operations.

6.1 Suppressing Output

MATLAB normally prints the result of every command. To prevent this, append a **semicolon (;)** at the end of the statement.

Examples:

```
x = 5;          % output suppressed
y = sqrt(10);   % output suppressed
z = x + y       % output shown because no semicolon
```

Suppressing output is useful when:

- Writing long scripts
- Creating large matrices
- Avoiding unnecessary clutter in the Command Window

[% is used for commenting purpose]

6.2 Accessing Help and Documentation

MATLAB provides extensive built-in documentation for functions and commands.

Key commands:

1. **help functionName**: Displays a brief description and basic usage in the Command Window. Example: `help log`
2. **doc functionName:** Opens the full documentation in the Help Browser, including examples, syntax, and explanations. Example: `doc plot`
3. **lookfor keyword:** Searches MATLAB documentation for a word or concept when you do not know the specific function name. Example: `lookfor cosine`

These tools help students learn MATLAB independently and troubleshoot functions they are unfamiliar with.

6.3 Input and Output Operations

This includes printing results to the screen and receiving data from the user.

6.3.1 Displaying Output

Use the disp() function to show messages or variable values.

```
disp("Welcome to MATLAB");
disp(x);
```

disp() is simpler than printing using formatted functions and is ideal for general script output.

6.3.2 Taking Input from the User

Use the input() function to read values typed in the Command Window.

Numeric Input:

```
n = input("Enter a number: ");
```

Text Input:

Use 's' to treat the input as a string:

```
name = input('Enter your name: ', 's');
```

7. Symbolic Operators in MATLAB

In MATLAB, **operators** are special symbols that perform mathematical or logical operations on variables and values. They are called **symbolic operators** because each operator is represented by a symbol (such as +, -, *, ==, etc.). Operators are essential for:

- Performing arithmetic calculations
- Comparing values
- Controlling program logic
- Operating on vectors and matrices
- Element-wise computations

MATLAB supports several categories of operators, each designed for a specific type of task.

7.1 Arithmetic Operators

Used for numeric calculations.

Table 1.5: Arithmetic Operators

Operator	Meaning	Example	Result
`+`	Addition	`3 + 2`	5
`-`	Subtraction	`7 - 5`	2
`*`	Matrix multiplication	`[1 2; 3 4] * [5; 6]`	Valid matrix product
`/`	Matrix right division	`A / B`	Equivalent to `A * inv(B)`
`\`	Matrix left division	`A \ B`	Equivalent to `inv(A) * B`
`^`	Matrix power	`A^2`	`A * A`
`.*`	Element-wise multiplication	`[1 2] .* [3 4]`	`[3 8]`
`./`	Element-wise division	`[6 8] ./ [2 4]`	`[3 2]`
`.\`	Element-wise left division	`[2 4] .\ [6 8]`	`[3 2]`

Operator	Meaning	Example	Result
`.^`	Element-wise power	`[2 3].^2`	`[4 9]`

7.2 Relational (Comparison) Operators

Used to compare values. Output is **logical** (1 = true, 0 = false).

Table 1.6: Relational (Comparison) Operators

Operator	Meaning	Example	Output
`==`	Equal to	`5 == 5`	1
`~=`	Not equal to	`5 ~= 3`	1
`>`	Greater than	`7 > 2`	1
`<`	Less than	`3 < 1`	0
`>=`	Greater or equal	`4 >= 4`	1
`<=`	Less or equal	`2 <= 1`	0

Relational operators are often used in:
- Conditions
- Loop controls
- Logical expressions

7.3 Logical Operators

Used to combine or invert logical expressions.

Table 1.7: Logical Operators

Operator	Meaning	Example	Output
`&`	Logical AND	`(5 > 3) & (2 < 1)`	0
`|`	Logical OR	`(5 > 3) | (2 < 1)`	1
`~`	Logical NOT	`~(5 > 3)`	0
`&&`	Short-circuit AND	`(3 > 5) && (10/0 > 1)`	0
`||`	Short-circuit NOT	`(5 > 2) || (10/0 > 1)`	1

Important:

- & and | work on vectors/matrices (element-wise).
- && and || only work on scalars.

7.4 Symbolic Math Operators (for Symbolic Toolbox)

Symbolic operators behave algebraically, not numerically:

Table 1.8: Symbolic Math Operators

Operator	Meaning	Example	Result
`+`	Symbolic addition	`x + y`	keeps as expression
`-`	Symbolic subtraction	`x - 3`	remains symbolic
`*`	Symbolic multiplication	`x*y`	algebraic product
`^`	Symbolic power	`x^3`	symbolic power

diff()	Differentiation	diff(x^2)	2*x
int()	Integration	int(x^2)	x^3/3
solve()	Solving equations	solve(x^2-4==0)	solutions symbolic

MATLAB Fundamentals - Cheat Sheet - Tools Course ETH Zürich

Basics

Workspace

`ans`	Most recent answer
`clc`	clear command window
`clear var`	clear variables Workspace
`clf`	Clear all plots
`close all`	Close all plots
`ctrl-c`	Kill the current calculation
`doc fun`	open documentation
`disp('text')`	Print text
`format` short—long	Set output display format
`help fun`	open in-line help
`load` filename {vars}	load variables from .mat file
`save` {-append} file {vars}	save var to file
`addpath` path	include path to ..
`iskeyword` arg	Check if arg is keyword
`% This is a comment`	Comments
`...`	connect lines (with break)
`";" (after command)`	suppresses output
`scriptname`	runs scriptname.m
`tic, toc`	start and stop timer
`ver`	List of installed toolboxes
MATLAB Documentation:	mathworks.com/help/matlab/

Defining and Changing Variables

`a = 5`	Define variable a to be 5
`A = [1, 2, 3, 4;`	Set A to be a 3×4 matrix
`5, 6, 7, 8;`	"," separates columns
`9, 10, 11, 12]`	";" separates rows
`[A,B], horzcat(A,B)`	Concatenate arrays horizontally
`[A;B], vertcat(A,B)`	Concatenate arrays vertically
`x(2) = 7`	Change 2nd element of x to 7
`A(2,1) = 0`	Change $A_{2,1}$ to 0
`x(2:12)`	The 2nd to the 12th elem. of x
`x(1:3:end)`	Every 3rd elem. of x (1st to last)
`x(x>6)`	List elements > 6.
`x(x>8)=8`	change elements using condition
`A(4,:)`	Get the 4th row of A
`A(:,3)`	Get the 3rd column of A
`A(6, 1:3)`	Get 1st to 3rd elem in 6th row
`zeros(9, 5)`	Make a 9×5 matrix of zeros
`ones(9, 5)`	Make a 9×5 matrix of ones
`eye(7)`	Make a 7×7 identity matrix
`diag(x)`	Create diagonal matrix
`diag(A)`	Get diagonal elements of matrix
`meshgrid(x)`	2-D and 3-D grids
`7:15`	Row vector of $7, 8, \dots, 14, 15$
`a:ds:b`	lin. spaced vector with spacing ds
`linspace(1,20,35)`	Lin. spaced vector (35 elements)
`logspace(1, 1e5, 50)`	Log. spaced vector (50 elements)

Arithmetics

`+, -`	Addition, Subtraction (elementwise)
`A*B`	Matrix multiplication
`A.*B`	elementwise multiplication
`A./B`	elementwise division
`B.\A`	Left array division
`/`	Solve $xA = B$ for x
`\`	Solve $Ax = B$ for x
`A^n`	normal/(square) matrix power
`A.^n`	Elementwise power of A
`sum(X)`	Sum of elements (along columns)
`prod(X)`	Product of elements (along columns)

Elementary Functions

`sin(A)`	Sine of argument in radians
`sind(A)`	Sine of argument in degrees
`asin(A)`	Inverse sine in radians
`sinh(A)`	Hyperbolic sine
there are analogous elementwise trigonometric functions for `cos`, `tan` and `cot`	
`abs(A)`	Compute $\lvert x \rvert$
`sqrt(x)`	Compute $\sqrt{x}$
`log(x)`	Compute $\ln(x)$
`log10(x)`	Compute $\log_{10}(x)$
`sign(x)`	sign of x
`exp(x)`	exponential of x

Complex Numbers

`abs(z)`	Absolute value and complex magnitude
`angle(z)`	Phase angle
`complex(a,b)`	Create complex numbers
`conj(z)`	Elementwise complex conjugate
`i or j`	Imaginary unit
`imag(z)`	Imaginary part of complex number
`isreal(z)`	Determine whether array is real
`real(z)`	Real part of complex number
`ctranspose(Z)`	Complex conjugate transpose

Constants

`pi`	$\pi = 3.141592653589793$
`NaN`	Not a number (i.e. 0/0)
`Inf`	Infinity
`eps`	Floating-point relative accuracy
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

Numerics and Linear Algebra

Numerical Integration and Differentiation

`integral(f,a,b)`	Numerical integration
`integral2(f,a,b,c,d)`	2D num. integration
`integral3(f,a,b,..,r,s)`	3D num. integration
`trapz(x,y)`	Trapezoidal integration
`cumtrapz(x,y)`	Cumulative trapez integration
`diff(X)`	Differences (along columns)
`gradient(X)`	Numerical gradient

Matrix Functions/ Linear Algebra

`A'`	Transpose of matrix or vector
`inv(A)`	inverse of A (use with care!)
`det(A)`	determinant of A
`eig(A),eigs(A)`	eigenvalues of A (subset)
`cross(A,B)`	Cross product
`dot(A,B)`	Dot product
`kron(A,B)`	Kronecker tensor product
`norm(x)`	Vector and matrix norms
`linsolve(A,B)`	Solve linear system of equations
`rank(A)`	Rank of matrix
`trace(A)`	Sum of diagonal elements
`curl(X,Y,Z,U,V,W)`	Curl and angular velocity
`divergence(X,..,W)`	Compute divergence of vector field
`null(A)`	Null space of matrix
`orth(A)`	Orthonormal basis for matrix range
`mldivide(A,B)`	Solve linear system $Ax = B$ for x
`mrdivide(B,A)`	Solve linear system $xA = B$ for x
`decomposition(A)`	Matrix decomposition
`lsqminnorm(A,B)`	Least-squares solution to linear eq.
`rref(A)`	Reduced row echelon form
`balance(A)`	Diagonal scaling (improve eig. vec.)
`svd(A)`	Singular value decomposition
`gsvd(A,B)`	Generalized svd
`chol(A)`	Cholesky factorization

Matrix manipulation

`cat(dim,A,B)`	Concatenate arrays
`ndims(A)`	Number of array dimensions
`flip(A)`	Flip order of elements
`fliplr(A)`	Flip array left to right
`flipud(A)`	Flip array up to down
`squeeze(A)`	Remove dimensions of length 1
`reshape(A,sz)`	Reshape array
`size(A)`	size of A
`sort(A)`	Sort array elements
`sortrows(A)`	Sort rows of matrix or table
`length(A)`	Length of largest array dimension

Graphics

Plotting

`plot(x,y)`	Plot y vs. x
`axis equal`	Scale axes equally
`title('A Title')`	Add title to the plot
`xlabel('x axis')`	Add label to the x axis
`ylabel('y axis')`	Add label to the y axis
`legend('foo', 'bar')`	Label 2 curves for the plot
`grid`	Add a grid to the plot
`hold on / off`	Multiple plots on single figure
`xlim /ylim / zlim`	get or set axes range
`figure`	Start a new plot

Plot types

plot, semilogx, streamline, histogram, plot3, semilogy, surf, pie, stairs, polarplot, mesh, scatter, errorbar, contour, image, scatterhist, stackedplot, quiver, bar, plotmatrix, loglog, quiver3, stem, heatmap

Plot gallery: mathworks.com/products/matlab/plot-gallery

Programming methods

Functions

```
% defined in m-file
% File must have the same name as the function
function output = addNumbers(x, y)
    output = x + y; %multiple or var nr of args possible
end
```

Anonymous Functions

```
% defined via function handles
f = @(x) cos(x.^2)./(3*x);
```

Relational and logical operations

==	Check equality	~=	Check inequality
>	greater than	>=	greater or equal to
<	less than	<=	less or equal to
&, &&	logical AND	~	logical NOT
|, ||	logical OR	xor	logical exclusive-OR

if, elseif Conditions

```
if n<10
    disp('n smaller 10')
elseif n<20
    disp('n between 10 and 20')
else
    disp('n larger than 20')
end % control structures terminate with end
```

Switch Case

```
n = input('Enter a number: ');
switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case {1,2,3} %check three cases together
        disp('positive one')
    otherwise
        disp('other value')
end % control structures terminate with end
```

For-Loop

```
%loop a specific number of times, and keep track of each ...
    iteration with an incrementing index variable
%parfor might be used to parallelize the execution
for i = 1:3
        disp('cool');     % comment with some LaTeX in it: πx²
end % control structures terminate with end
```

While-Loop

```
%loops as long as a condition remains true
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    n = n + 1;
    nFactorial = nFactorial * n;
end % control structures terminate with end
```

Further programming commands

`break`	exit the current loop (combine with `if`)
`continue`	go to next iteration (combine with `if`)
`try, catch`	Execute statements and catch errors

Special Topics

Polynomials

`poly(x)`	Polynomial with roots x
`poly(A)`	Characteristic polynomial of matrix
`polyeig(x)`	Polynomial eigenvalue problem
`polyfit(x,y,d)`	Polynomial curve fitting
`residue(b,a)`	Partial fraction expansion/decomposition
`roots(x)`	Polynomial roots
`polyval(p,x)`	Evaluate poly p at points x
`conv(u,v)`	Convolution and polynomial multiplication
`deconv(u,v)`	Deconvolution and polynomial division
`polyint(p,k)`	Polynomial integration
`polyder(p)`	Polynomial differentiation

Interpolation and fitting

`interp1(x,v,xq)`	1-D data interpolation (table lookup)
`interp2(X,Y,V,Xq,Yq)`	2D interpolation for meshgrid data
`interp3(X,..V,..Zq)`	3D interpolation for meshgrid data
`pchip(x,v,xq)`	Piecew. cubic Hermite poly interpol
`spline(x,v,xq)`	Cubic spline data interpolation
`ppval(pp,xq)`	Evaluate piecewise polynomial
`mkpp(breaks,coeffs)`	Make piecewise polynomial
`unmkpp(pp)`	Extract piecewise polynomial details

Differential equations

`ode45(ode,tspan,y0)`	Solve system of nonstiff ODE
`ode15s(ode,tspan,y0)`	Solve system of stiff ODE
`pdepe(m,pde,ic,bc,xm,ts)`	Solve 1D PDEs
`pdeval(m,xmesh,usol,xq)`	Interpolate num. PDE solution

Optimization

`fminbnd(fun,x1,x2)`	Find minimum of fun(x) in $[x_1, x_2]$
`fminsearch(fun,x0)`	Find minimum of function
`lsqnonneg(C,d)`	Solve non-neg. lin. least-squares prob.
`fzero(fun,x0)`	Root of nonlinear function
`optimget(opt,'par')`	Optimization options values
`optimset('opt',val)`	Define optimization options

Descriptive Statistics

`bounds(A)`	Smallest and largest elements
`max(A)`	Maximum elements of an array
`min(A)`	Minimum elements of an array
`mode(A)`	Most frequent values in array
`mean(A)`	Average or mean value of array
`median(A)`	Median value of array
`std(A)`	Standard deviation
`var(A)`	Variance
`hist(X)`	calculate and plot histogram
`corrcoef(A)`	Correlation coefficients
`cov(A)`	Covariance
`xcorr(x,y)`	Cross-correlation
`xcov(x,y)`	Cross-covariance
`rand`	Uniformly distributed random numbers
`randn`	Normally distributed random numbers
`randi`	Uniformly distributed pseudorandom integers

further functions: `movmax`, `movmin`, `cummax`, `cummin`, `movprod`, `movsum`, `cumsum`, `cumprod`, `movmean`, `movmedian`, `movstd`, `movvar`.

Discrete Math

`factor(n)`	Prime factors
`factorial(n`	Factorial of input
`gcd(n,m)`	Greatest common divisor
`lcm(n,m)`	least common multiple
`mod(a,m)`	Remainder after division (modulo operation)
`ceil(X)`	Round toward positive infinity
`fix(X)`	Round toward zero
`floor(X)`	Round toward negative infinity
`round(X)`	Round to nearest decimal or integer

Experiment 2: Defining Functions, Surface Plotting, and Debugging Codes

1. Objective of the Experiment

After completing this experiment, students will be able to:

1. Define and use MATLAB functions with single and multiple inputs/outputs.
2. Understand the relationship between scripts, functions, and the MATLAB workspace.
3. Generate surface plots of functions of two variables using `meshgrid`, `mesh`, and `surf`.
4. Use logical expressions, conditional statements, and loops to control program flow.
5. Identify and debug syntax, runtime, and logical errors using MATLAB debugging tools.

2. Introduction

This experiment focuses on developing **structured MATLAB programs** by introducing students to functions, control statements, loops, surface plotting, and debugging techniques. This experiment moves toward **program design and logical flow control**. Students will learn how to:

- Write **user-defined functions** for reusable and modular code
- Use **conditional statements** to make decisions within programs
- Apply **loops** to perform repetitive computations efficiently
- Generate **surface plots** to visualize functions of two variables
- Identify and correct **errors** using MATLAB debugging tools

All concepts in this experiment rely on previously learned fundamentals such as variables, matrices, operators, and built-in functions, and extend them into practical problem-solving and visualization tasks commonly used in engineering and scientific applications.

3. MATLAB Functions

A **function** in MATLAB is a self-contained block of code designed to perform a specific task. Unlike scripts, functions operate in a **controlled and isolated environment**, making programs more reliable, reusable, and easier to debug. MATLAB executes commands interactively in the Command Window or sequentially in scripts. A **function** extends this idea by:

- Encapsulating code into a reusable block
- Using a **local workspace** (separate from the base workspace)
- Accepting inputs and returning outputs

A function must be saved in a **separate `.m` file** with the **same name as the function**. Once defined, a function can be called repeatedly with different inputs.

3.1 Function Structure and Execution Flow

General Syntax

```
function [output1, output2, ...] = function_name(input1, input2, ...)
    % Description of the function
    Statements;
end
```

When a MATLAB function is called, MATLAB first identifies the function file and reads the function definition. The input values provided by the user are then passed to the function, and MATLAB creates a separate local workspace for that function. The statements inside the function execute sequentially using these inputs. Once execution is complete, the output variables are computed and returned to the calling program. Finally, the local workspace is cleared, ensuring that variables inside the function do not affect the base workspace.

Function File Naming Rule

- The **file name must exactly match the function name**. If names do not match, **MATLAB cannot find the function**
- Inputs and outputs are optional

Example:

```
function [y] = square(x)
    y = x^2;
end
```

Must be saved as: **square.m**

Inputs to a Function:

Inputs allow a function to work with different data. Inputs can be:

- Scalars
- Vectors
- Matrices
- Strings

MATLAB does **not** require type declaration.

Example [write the function in the editor section]

```
function y = cube(x)
    y = x^3;
end
```

Call: **[call the function in the command window]**

```
a = cube(4)
a =
    64
```

Why Inputs Are Optional

A function may not require input arguments if:

- It performs a fixed task
- It uses internally defined values
- It relies on built-in constants or functions

Example (no input):

```
Editor Window:

function greet()
    disp('Welcome to MATLAB');
end

Command Window:

Greet ()

Output:

'Welcome to MATLAB'
```

Here, the function always performs the same action, so no input is required.

Function with Inputs but No Outputs:

```
function check_even(n)
    if mod(n,2)==0
        disp('Even number');
    else
        disp('Odd number');
    end
end
```

<u>Outputs from a Function:</u>

Functions can return:

- One output
- Multiple outputs
- No output (procedural functions)

Single Output

```
Editor Window:

function y = square(x)
y = x^2;
end

Command Window:

a =square(2)

Output:

a =
    4
```

Multiple Outputs

Example 1:

```
function [sumVal, prodVal] = operations(a, b)
    sumVal = a + b;
    prodVal = a * b;
end
```

Call on command window:

```
[s, p] = operations(3, 4)
```

```
Output:
s =
    7
P =
    12
```

Example 2:

```
function [area, circumference] = circle_properties(radius)
    area = pi * radius^2;
    circumference = 2 * pi * radius;

end
Call:
[a, c] = circle_properties(3)
disp(['Area: ', num2str(a), ', Circumference: ', num2str(c)])
```

This demonstrates MATLAB's ability to return **multiple variables**.

Why Outputs Are Optional

A function may not return outputs if:

- Its purpose is to display results

- It performs actions such as plotting or saving data
- It modifies files or figures rather than producing numeric values

Example (no output):

```
function plot_circle(r)
    theta = linspace(0,2*pi,100);
    x = r*cos(theta);
    y = r*sin(theta);
    plot(x,y)
end
```

The function performs a task (plotting) but does not return a value.

Function with Outputs but No Inputs

```
function t = current_time()
    t = datetime('now');
end
```

After saving the function, press Run and you will see current time in the command window.

3.2 Practice problem – Circle Area Function

```
function area = circle_area(radius)
    % circle_area calculates the area of a circle
    area = pi * radius^2;
end

Call from Command Window or script:

r = 5;
a = circle_area(r);
disp(['Area of the circle: ', num2str(a)]);
```

Try creating functions for volume/ area functions of different shapes.

3.3. Built-In Functions

MATLAB includes a large library of built-in functions that operate on scalars, vectors, and matrices.

Table 3.1: Some Built-In Functions

Function	**Description**	**Example**	**Output**
`sqrt()`	Square root	`sqrt(16)`	4
`sum()`	Sum of elements	`sum([1 2 3])`	6

Function	Description	Example	Output
mean()	Mean value	mean([6 8 7])	7
max()	Maximum value	max([1 5 3])	5
min()	Minimum value	min([2 1 7])	1
size()	Matrix dimensions	size(A)	m n
linspace()	Linear spacing	linspace(0,10,5)	0 2.5 5 7.5 10
meshgrid()	Grid generation	[X,Y]=meshgrid(x,y)	
disp()	Display output	disp(x)	-

3.4 Script vs Function

Table 3.2: Difference between a script and a function

Feature	Script	Function
Inputs	Not explicit	Explicit
Outputs	Not returned	Returned
Workspace	Base workspace	Local workspace
Reusability	Low	High
Debugging	Limited	Structured

Example:

```
x = 10;
y = testFunc(x);
disp(['Value of x in script: ', num2str(x)])
disp(['Value of y returned by function: ', num2str(y)])
function y = testFunc(x)
    x = x + 5;
    y = x;
end
```

After execution: Base x remains 10 but y becomes 15. Function x is destroyed as it was stored in Local Workspace of function, it cannot overwrite the value of x which has been stored in Base Workspace.

4. Conditional Statements

Conditional statements allow MATLAB programs to **make decisions** and execute different blocks of code depending on whether a given condition is true or false. They are essential for implementing logic, handling different cases, and controlling program flow in real-world engineering problems. In MATLAB, the primary conditional construct is the if–elseif–else statement. A conditional statement:

- Evaluates a logical expression
- Executes code only if a condition is satisfied
- Uses relational and logical operators
- Enables decision-making in programs

4.1 General Syntax

The `if` Statement:

```
if condition
    statements;
end
```

How it works:

If condition is true, the statements inside the if block are executed. If false, MATLAB skips the block.

Example: [write the code in command window]

```
x = 10;
if x > 5
    disp('x is greater than 5');
end
```

The `if-else` Statement

```
if condition
    statements
else
    statements
end
```

Example: [write the code in command window]

```
num = 7;
if mod(num,2) == 0
    disp('Number is even');
else
    disp('Number is odd');
end
```

The `if-elseif-else` Statement:

Used when multiple conditions must be checked.

```
if condition1
    statements;
elseif condition2
    statements;
else
    statements;
end
```

Example:

```
x = -3;
if x > 0
    disp('Positive number');
elseif x < 0
    disp('Negative number');
else
    disp('Zero');
end
```

MATLAB evaluates conditions top to bottom and executes the first true condition only.

4.2 Conditions Using Logical Operators

AND condition

```
x = 8;
if (x > 5) && (x < 10)
    disp('x is between 5 and 10');
end
```

OR condition

```
x=8;
if (x < 0) || (x > 100)
    disp('x is outside the valid range');
end
```

NOT condition

```
x=8;
if ~(x == 0)
    disp('x is not zero');
end
```

Element-Wise AND (&)

```
A = [1 0 1 1];
B = [1 1 0 1];
C = A & B

Output:

1   0   0   1
```

Explanation:

- Each element is compared individually
- Result is `1` only where both elements are nonzero

Element-Wise OR (|)

```
A = [0 1 0];
B = [0 0 1];
C = A | B

Output:

0   1   1
```

Explanation: Result is `1` if **either element** is nonzero

Logical Indexing with Element-Wise Operators

```
v = [2 5 8 1 9];
selected = (v > 3) & (v < 9)

Output:

0   1   1   0   0
```

To extract values:

```
v(selected)

Output:

5   8
```

4.3 Nested Conditional Statements

An `if` statement can exist inside another `if`.

Example:

```
x = 15;
if x > 0
    if x < 10
        disp('Single-digit positive number');
    else
        disp('Positive number with two or more digits');
    end
end
```

4.4 Conditional Logic Inside Functions

Conditional statements are commonly used to validate input or select output behavior**.**

```
function [category] = classify_number(x)
    if x > 0
        category = 'Positive';
    elseif x < 0
        category = 'Negative';
    else
        category = 'Zero';
    end
end
```

5. Loops

Loops are used in MATLAB to repeat a block of code multiple times. They are essential when performing repetitive calculations, iterating over data, and constructing algorithms where the number of repetitions is known or condition-dependent.

MATLAB primarily uses:

- for loops
- while loops

Loops allow you to:

- Automate repetitive tasks
- Process vectors and matrices element by element
- Perform iterative numerical calculations
- Control program flow with conditions

Without loops, repeated statements would need to be written multiple times, making code inefficient and error-prone.

5.1 `for` Loop Syntax

```
for index = start_value : step_size : end_value
    statements
end
```

Here, index is the loop counter which can be expressed as i. The loop runs from start_value to end_value. The step_size is optional (default is 1).

Simple Example

```
for i = 1:5
    disp(i)
end

Output:

1
2
3
4
5
```

<u>How a `for` Loop Executes</u>

1. MATLAB assigns the first value to the loop variable.
2. Executes the loop body.
3. Increments the loop variable.
4. Repeats until the end value is reached.
5. Exits the loop.

If the loop range is empty, the loop body does **not execute at all**.

<u>Using Step Size in `for` Loops</u>

```
for i = 0:2:10
    disp(i)
end

Output:

0
2
4
6
8
10
```

Example: Sum of First 5 Natural Numbers

```
sum_val = 0;
for i = 1:5
    sum_val = sum_val + i;
end
disp(['Sum = ', num2str(sum_val)]);
```

5.2 Loops with Matrices

```
A = [1 2; 3 4];

for i = 1:size(A,1)
    for j = 1:size(A,2)
        disp(A(i,j))
    end
end
```

This uses **nested loops** to access each matrix element.

5.3 Nested Loops

A loop inside another loop is called a **nested loop**.

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

Nested loops are commonly used for:

- Matrix operations
- Surface plot calculations
- Algorithm implementation

5.4 Nested Loops with Conditions

Loops often contain conditions to control behavior.

```
for i = 1:10
    if mod(i,2) == 0
        disp([num2str(i), ' is even']);
    end
end
```

Only even numbers are displayed.

5.5 Using Loops in Functions

Loops are frequently used inside functions.

```
function s = sum_n(n)
    s = 0;
    for i = 1:n
        s = s + i;
    end
end
```

This function computes the **sum of the first n natural numbers**.

5.6 Loop Control Statements

1. `break`

Exits the loop immediately.

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

Output:
1
2
3
4
```

2. `continue`

Skips the current iteration.

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

6. Surface Plotting

Surface plotting in MATLAB is used to **visualize how a function of two variables behaves in three-dimensional space**. It is commonly applied in engineering and scientific problems where a dependent variable varies with respect to two independent variables.

A **surface plot** represents a function:

$$z = f(x,y)$$

where:

- `x` and `y` are independent variables (horizontal axes)
- `z` is the dependent variable (vertical axis)

Each point `(x, y)` has a corresponding height `z`.

Surface plots help students:

- Understand the shape and behavior of mathematical functions
- Visualize peaks, valleys, and curvature
- Analyze relationships between variables
- Interpret engineering models graphically

6.1 Basic Steps for Surface Plotting

Surface plotting in MATLAB follows four simple steps:

1. Create values for `x` and `y`
2. Generate a grid using `meshgrid`
3. Compute `z` values
4. Plot the surface using `surf`

Example:

```
[x, y] = meshgrid(-5:1:5, -5:1:5);
z = x.^2 + y.^2;
surf(x, y, z)
xlabel('X-axis')
ylabel('Y-axis')
zlabel('Z-axis')
title('Surface Plot of z = x^2 + y^2')
```

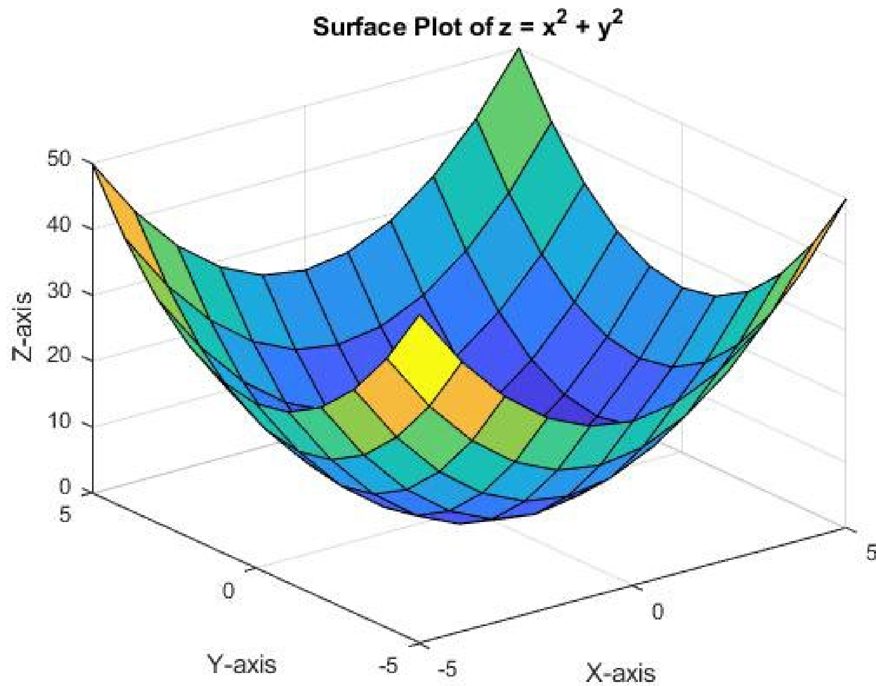


Figure 2.1: Surface Plot of z = x^2 + y^2

Meshgrid:

- Converts 1-D coordinate vectors into 2-D coordinate matrices
- Generates all possible (x, y) coordinate combinations
- Repeats the x-vector across rows
- Repeats the y-vector down columns
- Produces output matrices (X, Y) of equal size

Another Simple Example:

```
[x, y] = meshgrid(-pi:0.5:pi, -pi:0.5:pi);
z = sin(x).*cos(y);

surf(x, y, z)
title('Surface Plot of z = sin(x)cos(y)')
```

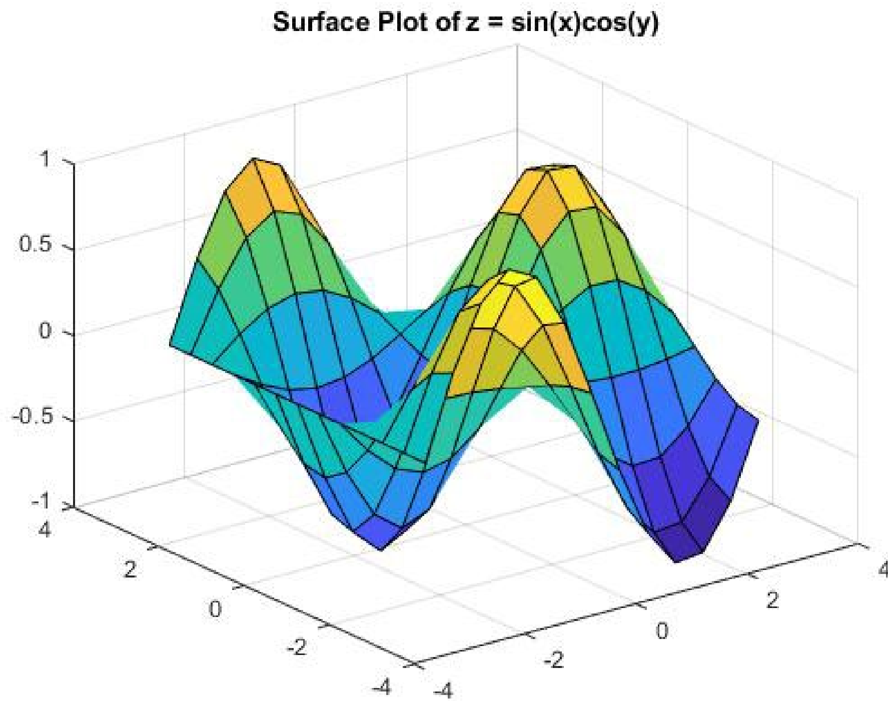


Figure 2.2: Surface Plot of z = sin(x)cos(y)

6.2 Surface Plot Using Loops

```
[X, Y] = meshgrid(-2:1:2, -2:1:2);
Z = zeros(size(X));

for i = 1:size(X,1)
    for j = 1:size(X,2)
        Z(i,j) = X(i,j)^2 + Y(i,j)^2;
    end
end

surf(X, Y, Z);
title('Surface Plot of Z(i,j) = X(i,j)^2 + Y(i,j)^2');
```

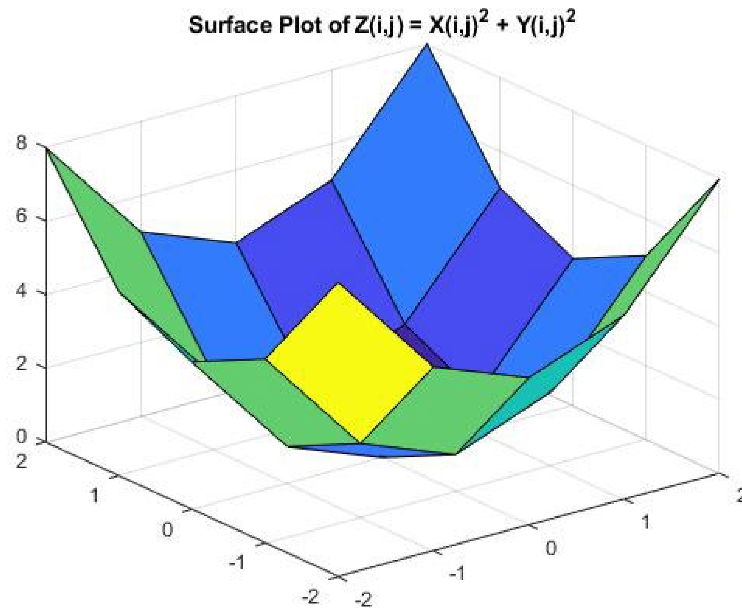


Figure 2.3: Surface Plot of Z(i,j) = X(i,j)^2 + Y(i,j)^2

7. Debugging MATLAB Codes

Debugging is the process of identifying, understanding, and correcting errors in a program so that it executes correctly and produces the intended results. Debugging is an essential skill for all MATLAB users, especially when working with functions, conditions, loops, and plots.

Types of Errors

- **Syntax errors:** Missing `end`, incorrect function definition
- **Runtime errors:** Dimension mismatch, division by zero
- **Logical errors:** Incorrect output despite correct syntax

Each type requires a different debugging approach.

- **Syntax Errors**

Syntax errors occur when MATLAB cannot understand the structure of the code. It occurs dure to:

- Missing `end`
- Misspelled keywords
- Incorrect use of brackets or parentheses

Example

```
if x > 5
    disp('x is greater than 5')
```

Error: Missing `end`

Fix

```
if x > 5
    disp('x is greater than 5')
end
```

MATLAB usually highlights syntax errors and stops execution immediately.

• Runtime Errors

Runtime errors occur **during execution**, even though the code is syntactically correct. It is commonly caused due to:

- Division by zero
- Index out of bounds
- Dimension mismatch

Example

```
x = 0;
y = 10 / x;
```

Error: Division by zero

• Logical Errors

Logical errors occur when:

- Code runs without error
- Output is incorrect

Example

```
x = 5;
if x > 10
    disp('x is greater than 10')
end
```

Issue: Condition is incorrect, so output never appears.

7.1 General Programming Errors

7.1.1 Overwriting Built-in Functions

```
sum = 10;
```

Issue: `Shadows built-in sum()`

Fix: Avoid using built-in names as variables.

7.1.2 Forgetting Element-wise Operators

```
x = 1:5;
y = x^2;    % ❌ Error
```

Fix:

```
y = x.^2;
```

7.1.3 Dimension Mismatch

```
A = [1 2 3];
B = [1; 2; 3];
C = A + B;    % ❌ Error
```

Fix:

```
C = A + B';  % Match dimensions
```

7.2 MATLAB Debugging Tools

7.2.1 Error Messages

Always read the error message:

- Error type
- Line number
- Cause

7.2.1 Using disp and fprintf

```
disp(x)
fprintf('Value of i = %d\n', i)
```

These are used to inspect variable values.

7.2.3 Breakpoints

- Click next to line numbers

- Pause execution
- Inspect variables step-by-step

Debugging Strategy (Recommended Workflow)

1. Read the error message carefully
2. Check syntax (end, brackets, operators)
3. Verify conditions and loop logic
4. Inspect variable sizes using size() or whos
5. Use breakpoints to step through execution
6. Test code with simple inputs

Table 7.1: Common Errors in Conditional Statements (MATLAB)

Error Type	**Incorrect Code Example**	**Cause**	**Correct Code / Fix**
Assignment instead of comparison	`if x = 5`	`=` assigns value instead of comparing	`if x == 5`
Missing `end`	`if x > 0 disp(x)`	Incomplete block	Add `end`
Using `&&` with vectors	`if v > 1 && v < 5`	`&&` works only with scalars	`if all(v > 1 & v < 5)`
Incorrect logical operator	`if x > 0 & x < 10`	Works but poor practice for scalars	Use `&&`
Missing parentheses	`if x > 0 && x < 10`	Logical grouping unclear	`if (x > 0) && (x < 10)`
Condition always false	`if x > 10` (when x = 5)	Wrong logic	Verify condition
Vector condition in `if`	`if v > 0`	`if` expects scalar logical	Use `any()` or `all()`
Using strings instead of chars	`if x == "yes"`	Type mismatch	Use consistent types

Table 7.2: Common Errors in Loops (MATLAB)

Error Type	**Incorrect Code Example**	**Cause**	**Correct Code / Fix**
Infinite loop	`while x < 10` (no update)	Loop variable never changes	Update loop variable
Loop never executes	`for i = 5:1`	Invalid range	`for i = 5:-1:1`
Missing `end`	`for i = 1:5 disp(i)`	Incomplete loop block	Add `end`
Index out of bounds	`v(i)` where `i > length(v)`	Invalid indexing	Use `1:length(v)`
No preallocation	`A(i)=i`	Performance issue	Preallocate array
Wrong loop variable used	`for i=1:5 disp(j)`	Typo or logic error	Use correct variable
Nested loop logic error	Reused loop index	Index overwritten	Use unique indices

Error Type	Incorrect Code Example	Cause	Correct Code / Fix
Using vector as loop limit	`for i = v` unintentionally	Misunderstanding syntax	Use indices instead

9. Practice Problems:

Problem 1: Heat Distribution on a Plate

Description:

Simulate a simple 2D heat distribution on a square plate. The temperature T(x,y) at each point is defined as:

$$T(x,y) = \begin{cases} 100 - (x^2 + y^2), & if\ x^2 + y^2 \leq 50 \\ 50, & otherwise \end{cases}$$

Hints:

1. Define a **function** that calculates T given x and y.
2. Use **nested for loops** to calculate T for x and y in the range -10 to 10 with step 1.
3. Use **conditional statements** to apply the correct formula based on distance from the center.
4. Create a **surface plot** to visualize temperature distribution.

Problem 2: Parabolic Bowl with Condition

Description:
Generate a surface representing a parabolic bowl:

$$Z(x,y) = \begin{cases} x^2 + y^2, & if\ x^2 + y^2 \leq 25 \\ 0, & otherwise \end{cases}$$

Hints:

1. Implement a **function** `parabolicBowl(x, y)` for `z`.
2. Use **nested for loops** to compute `z` for `x` and `y` from -10 to 10 with step 0.5.
3. Use **conditional statements** inside the function.
4. Create a **surface plot** with axis labels and title.

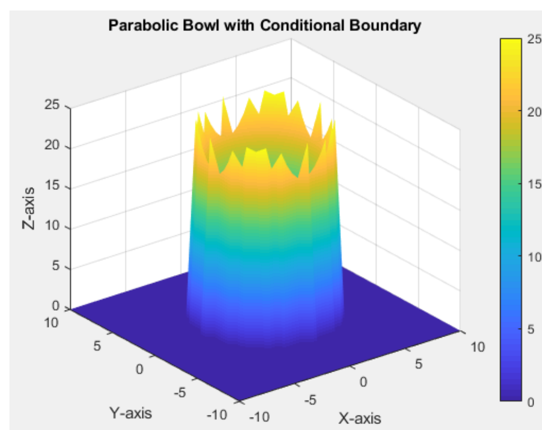
Solution (Problem 2):

Create a function in a new script/editor. Save it as per function name.

```
function z = parabolicBowl(x, y)
    if (x^2 + y^2) <= 25
        z = x^2 + y^2;
    else
        z = 0;
    end
end
```

Next, you can write your code at your editor or your command window:

```
clc,clear;
x = -10:0.5:10;
y = -10:0.5:10;
Z = zeros(length(x), length(y));    % Preallocation of Z matrix
for i = 1:length(x)
    for j = 1:length(y)
        Z(i, j) = parabolicBowl(x(i), y(j));
    end
end
[X, Y] = meshgrid(x, y);
surf(X, Y, Z)
shading interp
xlabel('X-axis')
ylabel('Y-axis')
zlabel('Z-axis')
title('Parabolic Bowl with Conditional Boundary')
colorbar
```



Experiment 3: Introduction to Matrix and Array

1. Objective of the Experiment

After completing this experiment, students will be able to:

i. Differentiate between Matrix and Array Operations: Understand and apply the distinction between linear algebra-based matrix operations (e.g., *, /) and elementwise array operations using the dot operator (e.g., .*, ./).
ii. Perform Basic Arithmetic on Multidimensional Data: Execute fundamental calculations like addition, subtraction, and multiplication on vectors and matrices while ensuring dimensional compatibility.
iii. Utilize Built-in Constructor Functions: Generate specific data structures efficiently using standard commands such as zeros, ones, eye, and rand.
iv. Apply Scalar Expansion: Use scalar-to-array operations to automatically apply a single value across all elements of a matrix or vector.

2. Introduction

Array: All MATLAB variables are multidimensional arrays. Arrays can contain any data type (numeric, characters, etc.) and perform element-by-element computations.

Matrix: A 2D numeric array used specifically for linear algebra transformations. Operations follow standard mathematical matrix rules (e.g., inner dimensions must match for multiplication).

Note: All variables of all data types in MATLAB are multidimensional arrays. A vector is a one-dimensional array and a matrix is a two-dimensional array.

3. One Dimensional Numeric Array/ Vector:

a. Row Vector and Column Vector

i. Row Vectors: Row vectors are created by enclosing the set of elements in square brackets, using space or comma to delimit the elements. Example:
`RV = [9 8 7 6 5];`

ii. Column Vectors: Column vectors are created by enclosing the set of elements in square brackets, using semicolon to delimit the elements. Example:
`CV = [1; 8; 10; 10; 15];`

b. Uniformly Spaced Matrix

i. To create a vector v with the first element f, last element l, and the difference between elements is any real number n, the syntax is `v = [f: n: l]`

c. Referencing the Elements of a Vector

One can reference one or more of the elements of a vector in several ways. The ith component of a vector CV is referred as CV (i). Example:

```
CV = [9; 8 ;7 ;6; 5];
CV (3)
```

```
Output:
ans = 7
```

Again, reference a vector with a colon, such as CV (:), all the components of the vector are listed. However, one can select the range of elements from a vector. Example:

```
RV = [9 8 7 6 5]; Sub_RV= RV (2:4)
Output: Sub_RV = 8 7 6
```

d. Addition and Subtraction of Vectors

To add or subtract two vectors, both the operand vectors must be of same type and have same number of elements. Example:

```
A = [1, 2, 3, 4, 5]; B = [6, 7, 8, 9, 10]; C = A + B; D =
A - B; disp(C); disp(D);
```

e. **Scalar Multiplication of a Vector** The multiplication of a vector by a number is called the scalar multiplication. Scalar multiplication produces a new vector of same type with each element of the original vector multiplied by the number. Example:

```
RV = [9 8 7 6 5]
MV=5*RV
```

Note: It is possible to perform all scalar operations on vectors. For example, one can add, subtract and divide a vector with a scalar quantity.

f. Transpose a Vector

The transpose operation changes a column vector into a row vector and vice versa. The transpose operation is represented by a single quote ('). Example:

```
TV= [1, 2, 3, 4, 5]'
```

If the code is executed, it will return the value of TV column-wise.

g. Append Vectors

One can append two vectors at end to end, or in separate row or column.

Example:

% just concatenate second vector in the same row at the end of the first vector

```
r1 = [ 1 2 3 4];
r2 = [5 6 7 8];
r = [r1, r2]
```

% just concatenate second vector in a new row at the end of the first vector

```
Row_Mat = [r1; r2]
```

% just concatenate second vector in the same column at the end of the first vector

```
c1 = [ 1; 2; 3; 4];
c2 = [5; 6; 7; 8];
c = [c1; c2]
```

% just concatenate second vector in a new column at the end of the first vector

```
Col_Mat = [c1, c2]
```

h. Magnitude of a Vector

Magnitude of a vector v with elements v1, v2, v3, , vn, is given by the equation:

$V = (v_1^2 + v_2^2 + v_3^2 + \dots\dots\dots + v_n^2)$

$|v| = \sqrt{V}$

Example:

```
v = [1: 2: 20];
sv = v.* v; %the vector with elements
% as square of v's elements
dp = sum(sv); % sum of squares -- the dot product
disp(sqrt(dp)); % magnitude
```

i. Vector Dot Product

Dot product of two vectors a = (a1, a2, a3) and b = (b1, b2, b3) is given by

$a.b = \sum(a_i.b_i)$

Example:

```
v1 = [2 3 4];
v2 = [1 2 3];
dp = dot (v1, v2)
```

4. Two-Dimensional Numeric Array/ Matrix:

a. Referencing the Elements of a Matrix

To reference an element in the mth row and n^{th} column, of a matrix Mat, then write: Mat(m, n); Example:

```
Mat=[1,2,3;4,5,6;7,8,9]
Mat =
     1     2     3
     4     5     6
     7     8     9
>> Mat (2,2)
ans =
     5
```

To reference all the elements in the mth column, type: Mat(:,m).

Example:

```
>> Mat(:,2)
ans =
     2
     5
     8
```

One can also select the elements in the mth through n^{th} columns or create a sub matrix, for this, write:

```
Mat(:,m:n);
>> Mat(:,2:3)
ans =
     2     3
     5     6
     8     9
```

To create a subset that contains the elements in the rth through sth rows and the elements in the mth through nth columns by writing:

```
Mat(r:s, m:n);
>> Mat(2:3,2:3)
ans =
     5     6
     8     9
```

b. Deleting a Row or a Column in a Matrix

To delete an entire row or column of a matrix by assigning an empty set of square braces [ ] to that row or column. Basically, [ ] denotes an empty array.

For example, let us delete the third row of Mat matrix:

```
Mat=[1,2,3;4,5,6;7,8,9]
Mat(3, :) = [ ]
```

After executing above statement and MATLAB return the following result:

```
Mat =
     1     2     3
     4     5     6
```

c. Concatenating Matrix

In MATLAB, the square brackets ([]) are used as the concatenation operator to combine matrices.

Horizontal concatenation combines matrices side-by-side using commas or spaces as separators, e.g., [A, B]. This requires the matrices to have the same number of rows.

Vertical concatenation stacks matrices on top of each other using semicolons as separators, e.g., [A; B]. This requires the matrices to have the same number of columns. Example:

```
a = [ 10 12 23; 14 8 6; 27 8 9];
b = [ 12 31 45; 8 0 -9; 45 2 11];
c = [a, b]
d = [a; b]
```

After executing above statement and MATLAB return the following result:

```
>> c =

    10    12    23    12    31    45
    14     8     6     8     0    -9
    27     8     9    45     2    11
d =

    10    12    23
    14     8     6
    27     8     9
    12    31    45
     8     0    -9
    45     2    11
```

d. Transpose a Matrix

In MATLAB, transpose (), ctranspose () or by using .' or ' will return the transpose of the matrix.

Example:

```
A = [1 2 3; 4 5 6; 7 8 9];
T = A .' or
T = A' or
T = transpose(A)
```

Then the output result will be:

```
>> A =
1 4 7
2 5 8
3 6 9
```

Note: .' performs a non-conjugate transpose (flips dimensions only), while ' performs a complex conjugate transpose (flips dimensions and negates imaginary parts).

e. Sparse Matrix

Sparse matrices are a type of matrix in which the majority of the elements are zero. To create a sparse matrix, syntax: S = sparse(A). The sparse () function automatically extracts the non-zero values and their corresponding row and column indices from the dense matrix A to create the sparse representation.

For Example:

```
>> A = [0, 0, 0, 0;0, 1, 0, 2; 0, 0, 3, 0;0, 4, 0, 5]
>> sparse_matrix = sparse(A)
A =
   0     0     0     0
   0     1     0     2
   0     0     3     0
   0     4     0     5
sparse_matrix =
Compressed Column Sparse (rows = 4, cols = 4, nnz = 5
[31%])
  (2, 2) -> 1
  (4, 2) -> 4
  (3, 3) -> 3
  (2, 4) -> 2
  (4, 4) -> 5
```

Now, creating a sparse matrix of nonzeros with a specified size involves providing the dimensions of the matrix along with the non-zero elements' locations and values like that:

```
num_rows = 5;
num_cols = 5;
nonzero_rows = [1, 2, 3, 4];
nonzero_cols = [2, 4, 1, 3];
nonzero_vals = [10, 20, 30, 40];
sparse_matrix = sparse (nonzero_rows, nonzero_cols,
nonzero_vals, num_rows, num_cols)
```

After pressing enter in the command window it will show:

```
sparse_matrix =
Compressed Column Sparse (rows = 5, cols = 5, nnz = 4
[16%])

  (3, 1) -> 30
  (1, 2) -> 10
  (4, 3) -> 40
  (2, 4) -> 20
```

5. Array versus Matrix Operations

The presence of a dot (.) distinguishes array operations from matrix operations.

Operation	Array Operation	Matrix Operation
Multiplication	`A .* B` or `times(A, B)`	`A * B`
Division	`A ./ B` or `rdivide(A, B)`	`A / B` or `A \ B`
Power	`A .^ 2`	`A ^ 2`
Transpose	`A.'`	`A'` (Complex conjugate transpose)

6. Array Function

Function Name	Syntax (Example)	Description
zeros	zeros (m, n)	Creates an array of size m x n filled entirely with zeros.
ones	ones (m, n)	Creates an array of size m x n filled entirely with ones.
eye	eye(n)	Creates an n x n identity matrix (1s on the diagonal, 0s elsewhere).
magic	magic(n)	Creates a magic square array
rand	rand (m, n)	Generates a matrix with random numbers from a uniform distribution (0 to 1).
randn	randn (m, n)	Generates a matrix with random numbers from a standard normal distribution.
size	size(A)	Returns the dimensions of matrix A as a row vector [m, n].
reshape	reshape (A, m, n)	Reconfigures an existing array into a new shape of m rows and n columns.
transpose	transpose(A)	Swaps the rows and columns of the matrix.

Function Name	Syntax (Example)	Description
max	max(A)	Returns the largest element found in the array or vector.
min	min(A)	Returns the smallest element found in the array or vector.
sum	sum(A)	Calculates the total sum of all elements in the array.
mean	mean(A)	Calculates the average (arithmetic mean) of the array elements.
diag	diag(v)	Creates a square matrix with vector v on the main diagonal.
unique	unique (A, setOrder)	Returns the unique values from the array, removing all duplicates. If setOrder will be 'Stable', then returns only unique elements without sorted.
sort	sort (A, direction)	Rearranges the elements of the array in ascending or descending order. To get array in descending order, direction will be 'descend'.

7. Compatible Arrays

In MATLAB, compatible arrays are considered compatible when they share the same data type and size, or if one of them is a scalar. During the execution of element-wise operations or functions in MATLAB, arrays with compatible sizes are automatically extended to align with the dimensions of each other.

Example:

```
>> matrix1 = [1, 2; 3, 4];
matrix2 = [5, 6; 7, 8];
add = matrix1 + matrix2
>> add =
    6      8
    10     12
```

```
>>scalar = 1;
result = add+scalar
>>
result =

    7      9
    11      13
```

8. Categorical Arrays

Categorical array is a data type in MATLAB that allows work with discrete data. Discrete data refers to a type of data that consists of distinct, separate values or categories. Discrete data can only take on specific, finite values, often in the form of whole numbers or distinct categories.

Categorical Arrays can be created in MATLAB by using –

- categorical () Function

  Example:

```
A = [3 2;3 3;3 2;2 1;3 2]
valueset = [1:3];
categorynames = {'poor' 'fair' 'good'};
B =
categorical (A, valueset, categorynames,
'Ordinal',true)
```

  If the code is executed, it will show:

```
A =

   3     2
   3     3
   3     2
   2     1
   2     2
B =

   5x2 categorical array

   good      fair
   good      good
   good      fair
   fair      poor
   good      fair
```

  Here, categorical function converts a numeric matrix A into a categorical matrix B by mapping the values 1, 2, and 3 to the labels 'poor', 'fair', and 'good'. By setting 'Ordinal', true, it establishes a logical ranking where 'poor' < 'fair' < 'good'.

- discretize () Function

  Example:

```
scores =
[68, 75, 82, 90, 55, 78, 92, 60, 88, 72];
edges = [0, 60, 80, 100];
categories = discretize (scores, edges)
```

On execution the output is as follows-

```
categories =
   2      2      3      3      1      2      3      2      3
2
```

In this example, the edges array defines the intervals for categorizing the scores. Scores below 60 will be categorized as "Low", scores between 60 and 80 as "Medium", and scores between 80 and 100 as "High".

9. Multidimensional Arrays

An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. To generate a multidimensional array, we first create a two-dimensional array and extend it. Suppose, A= [1 2 3; 4 5 6; 7 8 9]; The array A is a 3-by-3 array; we can add a third dimension to A, by providing the values like A(:, :, 2)= [10 11 12;13 14 15;16 17 18];

Example:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A(:,:,2) = [11 12 13; 14 15 16; 17 18 19]
A =
     1     2     3
     4     5     6
     7     8     9
A(:,:,1) =
     1     2     3
     4     5     6
     7     8     9
A(:,:,2) =
    11    12    13
    14    15    16
    17    18    19
```

To refer to an element or a subset of elements in a multidimensional array, one can use the same subscripting method as a 2D matrix by specifying an index for each dimension (e.g., A (row, col, page)).

10. Cell Arrays

In MATLAB, a cell array is a flexible data structure that allows you to store data of different types and sizes. This is different from regular arrays where the elements in the array have to be of the same data type. One can create cell array by –

- Using cell array operator { }

  Example:

```
>> C = {'2023-07-23',[10 20 30 40]}
C =
  1×2 cell array
    {'2023-07-23'} {[10 20 30 40]}
```

Now, type the following code:

```
>> C(3,:) = {'2023-07-25',[1 2 3 4]}
C =
  3×2 cell array
    {0×0 double   }     {0×0 double}
    {0×0 double   }     {0×0 double}
```

```
     {'2023-07-25'}     {[ 1  2  3  4]}
```

- Using cell () function

  Example:

```
>> C = cell(2, 3)
C =
2x3 cell array
{0x0 double}     {0x0 double}     {0x0 double}
{0x0 double}     {0x0 double}     {0x0 double}
```

```
>> C{1, 1} = 'Hello';
C{1, 2} = 50;
C{1, 3} = [1, 2, 3];
C{2, 1} = 3.14;
C{2, 2} = 'TEST';
C{2, 3} = magic(3);
```

```
>> C
C =
  2x3 cell array
   {'Hello' }     {[   50]}     {[    1  2  3]}
   {[3.1400]}     {'TEST'}     {3x3 double}
```

Problem 1: An HR department records daily attendance of 6 workers over 5 working days (1 = Present, 0 = Absent).

Tasks

1. Store attendance data in a 6×5 matrix.
2. Calculate: Total attendance of each worker, Daily workforce availability
3. Identify: Workers with attendance below 80%
4. Convert attendance data into percentage attendance using array operations.
5. Remove workers who do not meet attendance requirements.

Problem 2: A warehouse stores 5 products and records stock levels for 7 days.

Tasks

1. Create an inventory matrix (products × days).
2. Calculate: Average stock of each product, Minimum stock level
3. Identify: Products that fall below reorder level
4. Simulate a 10% demand surge using element-wise operations.

5. Flag critical inventory days.

Problem 3: A Project engineer manages 5 project activities. Time estimates are given as:

- Optimistic (O)
- Most likely (M)
- Pessimistic (P)

Tasks

1. Store time estimates in a 5×3 matrix.
2. Compute: Expected time using PERT formula

$$T_e = \frac{O + 4M + P}{6}$$

3. Identify: Activities exceeding average project duration
4. Sort activities based on expected time.
5. Store activity name and expected time in a cell array.

Experiments 4: Matrix operations and manipulation

1. Objective of the Experiment

After completing this experiment, students will be able to:

i. Calculate fundamental matrix properties like determinant, rank, and trace etc. in MATLAB.
ii. Compute the inverse of a matrix and understand when a matrix is singular.
iii. Solve systems of linear equations using both matrix inverse and the efficient backslash operator (\).

2. Introduction

In Industrial and Production Engineering, matrices are fundamental for modeling complex systems such as inventory flows, production scheduling, statistical analysis (regression), and finite element analysis. This experiment focuses on core matrix operations and manipulations provided by MATLAB, for computational efficiency.

In industrial environments, data matrices often need to be dynamically input by the user or loaded from files, and operations must be performed robustly. The key is ensuring matrices are compatible. This manual focuses on writing MATLAB code that automatically checks these constraints before executing the operation, providing user-friendly feedback. We use the input () function to interact with the user and the size () function within conditional blocks to manage compatibility.

Operation Type	**Operation**	**Compatibility Rule**	**MATLAB Check / Condition**
Basic Arithmetic	Addition (+)	Both matrices must have identical dimensions $(m \times n)$	all(size(A) == size(B))
	Subtraction (−)	Both matrices must have identical dimensions $(m \times n)$	all(size(A) == size(B))
Matrix Multiplication	Matrix multiplication (*)	Inner dimensions must match $(m \times n) \times (n \times p)$	size(A,2) == size(B,1)
Element-wise Operations	Element-wise multiplication (.*)	Matrices must have identical dimensions	all(size(A) == size(B))
	Element-wise division (./)	Matrices must have identical dimensions	all(size(A) == size(B))
	Element-wise power (.^)	Matrices must have identical dimensions	all(size(A) == size(B))

Operation Type	Operation	Compatibility Rule	MATLAB Check / Condition
Division	Right division (/)	A/B means A * inv(B) → B must be square	size(B,1) == size(B,2)
	Left division ()	A\B solves AX = B → A must be square	size(A,1) == size(A,2)
Transpose	Transpose (')	Always valid; converts *(m × n) → (n × m)*	size(A') == fliplr(size(A))
Inverse	Matrix inverse (inv)	Matrix must be square and non-singular	size(A,1) ==size(A,2) && det(A)≠0
Determinant	Determinant (det)	Matrix must be square	size(A,1) == size(A,2)
Rank	Rank (rank)	Defined for any matrix	Always valid
Trace	Trace (trace)	Matrix must be square	size(A,1) == size(A,2)
Concatenation	Horizontal concatenation [A B]	Same number of rows	size(A,1) == size(B,1)
	Vertical concatenation [A; B]	Same number of columns	size(A,2) == size(B,2)
Linear System	Solve AX = B	A must be square, rows(A) = rows(B)	size(A,1) ==size(A,2) && size(A,1) ==size(B,1)
Eigenvalues	Eigenvalues (eig)	Matrix must be square	size(A,1) == size(A,2)
SVD	Singular Value Decomposition (svd)	Works for any m × n matrix	Always valid

Problem 1:

An industrial decision-support system requires multiple matrix operations based on user input.

Write a MATLAB program that:

1. Dynamically takes matrix sizes
2. Allows the user to choose an operation:
    - Addition
    - Subtraction
    - Multiplication
    - Determinant
    - Inverse
    - Trace
    - Rank

3. Checks compatibility for the selected operation
4. Executes the operation if valid
5. Displays informative error messages otherwise

Problem 2:

An Industrial and Production Engineer is responsible for forecasting future product demand to support production planning and inventory control. Historical demand data is available over several time periods. Instead of using built-in regression functions, the engineer decides to develop a linear regression forecasting model using matrix manipulation.

Given

- A set of historical observations:
  - Time periods stored in vector x
  - Corresponding demand values stored in vector y
- Linear regression model:

$$y = \beta_0 + \beta_1 x$$

Tasks

(a) Construct the design matrix Xusing the time vector x.
(Hint: Include a column of ones for the intercept.)

(b) Derive the normal equation to estimate the regression coefficients β_0 and β_1 using matrix operations.

(c) Write a MATLAB program that:

1. Dynamically takes the number of observations
2. Accepts vectors x and y from the user
3. Computes the regression coefficients using
$$\beta = (X^T X)^{-1} X^T y$$
4. Predicts demand for a future time period entered by the user
5. Displays the regression equation and forecasted value

(d) Plot:

- Actual demand data
- Regression line
- Forecasted demand point

Experiment 5: Auto-Initialization, Indexing and Variables Manipulation

Objectives

After completing this experiment, students will be able to:

- Understand and apply auto-initialization techniques to dynamically grow arrays and matrices.
- Master advanced indexing techniques, including linear indexing, logical indexing, and colon notation for data extraction.
- Perform complex variable manipulation to reshape, flip, rotate, and sort data structures.
- Utilize built-in functions to modify matrix dimensions and combine multiple data sets.
- Analyze the memory implications of dynamic array growth versus pre-allocation.

Introduction

This experiment explores the deeper mechanics of how MATLAB handles data structures and memory through auto-initialization and precise variable manipulation. While previous experiments focused on basic array creation, this session moves toward sophisticated data handling necessary for large-scale data analysis. One will explore how MATLAB handles variables that change size on the fly and how to "slice and dice" data to get exactly what one need. Students will learn how to:

- Auto-initialize arrays by assigning values to indices beyond current dimensions and understand the resulting "zero-padding."
- Manipulate variables using functions like reshape, repmat, cat, and transpose to restructure data for specific mathematical operations.
- Apply advanced indexing to isolate specific sub-blocks of data or filter elements based on logical conditions.
- Optimize performance by comparing the efficiency of dynamic initialization against pre-allocation strategies (e.g., using zeros or ones).

These techniques are essential for managing the complex data arrays encountered in professional engineering environments, allowing for more flexible, efficient, and readable code.

Creating MATLAB variables

MATLAB variables are created with an assignment statement. The syntax of variable assignment is

```
variable name = a value (or an expression)
```

For example,
>> x = expression; where expression is a combination of numerical values, mathematical operators, variables, and function calls. On other words, expression can involve:

- manual entry
- built-in functions
- user-defined functions

Overwriting variable

Once a variable has been created, it can be reassigned.

```
>> t=5;
>> t=t+1

   t =6
```

Error messages

If we enter an expression incorrectly, MATLAB will return an error message. For example, in the following, we left out the * in the following expression

```
>> x=10;
>> 5x

??? 5x
```

Error: Unexpected MATLAB expression.

Making corrections

To make corrections, we can, of course retype the expressions. But if the expression is lengthy, we make more mistakes by typing a second time. A previously typed command can be recalled with the up-arrow key ↑. When the command is displayed at the command prompt, it can be modified if needed and executed.

Controlling the hierarchy of operations or precedence

Let's consider the previous arithmetic operation, but now we will include parentheses. For example, $1 + 2 \times 3$ will become $(1 + 2) \times 3$

```
>> (1+2)*3
ans = 9
```

By adding parentheses, these two expressions give different results: 7 and 9.

Therefore, to make the evaluation of expressions unambiguous, MATLAB has established a series of rules. The order in which the arithmetic operations are evaluated is given in Table, MATLAB arithmetic operators obey the same precedence rules as those in most computer programs. For operators of equal precedence, evaluation is from left to right.

Precedence Level	**Mathematical Operation**	**Description**
First	Parentheses	Expressions inside parentheses are evaluated first, starting from the innermost pair and proceeding outward.
Second	Exponentiation	All exponential operations are evaluated next, working from left to right.
Third	Multiplication and Division	Multiplication and division operations are evaluated from left to right.
Fourth	Addition and Subtraction	Addition and subtraction are evaluated last, working from left to right.

Controlling the appearance of floating-point number

By default, MATLAB displays numbers with four decimal place values. This is known as short format. However, if one wants more precision, he/she needs to use the format command.

```
format long

x = 7 + 10/3 + 5^1.2
```

MATLAB will execute the above statement and return the following result –

```
x = 17.2319816406394
```

Another example,

```
format short

x = 7 + 10/3 + 5^1.2
```

MATLAB will execute the above statement and return the following result –

```
x = 17.232
```

The format bank command rounds numbers to two decimal places. For example,

```
format bank

daily_wage = 177.45;
weekly_wage = daily_wage * 6
```

MATLAB will execute the above statement and return the following result –

```
weekly_wage = 1064.70
```

MATLAB displays large numbers using exponential notation.

The format short e command allows displaying in exponential form with four decimal places plus the exponent.

```
format short e

4.678 * 4.9
```

MATLAB will execute the above statement and return the following result –

```
ans = 2.2922e+01
```

The format long e command allows displaying in exponential form with four decimal places plus the exponent. For example,

```
format long e

x = pi
```

MATLAB will execute the above statement and return the following result –

```
x = 3.141592653589793e+00
```

The format rat command gives the closest rational expression resulting from a calculation. For example,

```
format rat

4.678 * 4.9
```

MATLAB will execute the above statement and return the following result –

```
ans = 34177/1491
```

Variable Assignment and Auto-Initialization

MATLAB allows variables to be created and initialized automatically without prior declaration or dimensioning.

- **Direct Assignment:** Simply assigning a value creates the variable. Example: x = 10; y = "Hello"; A = [1 2 3];
- **Automatic Growth:** Assigning a value to an index beyond the current array bounds automatically expands the array and fills missing elements with zeros.

  Example: If A = [1, 2], then A(5) = 10 results in A = [1, 2, 0, 0, 10].

- **Scalar Expansion:** Assigning a single scalar to a range of indices fills those indices with that value. Example: A(1:3) = 0;.

- **Dynamic typing:** A variable's type can change when a new value of a different type is assigned.

  Example:

  myVar = 10;   % myVar is a double
  myVar = "now text"; % myVar is now a string
- **Pre-allocation (for efficiency):** While not required for functionality, pre-allocating large arrays using functions like zeros, ones, or rand before a loop improves code performance by reserving memory in advance.

  ```
  % Efficient pre-allocation of a 1000-element array
  data = zeros(1000, 1);
  for i = 1:1000 data(i) = i * 2;

  end
  ```

- **Initialization Functions:** Use built-in functions for specific structures:

  - zeros(m, n): Creates an $m \times n$ matrix of zeros.

  - ones(m, n): Creates an $m \times n$ matrix of ones.

  - eye(n): Creates an $n \times n$ identity matrix.

myCellArray = cell(10, 1); % Preallocates a 10x1 cell array
myArray = zeros(100, 100); % Preallocates a 100x100 array of
zeros

Array Indexing

MATLAB uses **1-based indexing**, meaning the first element is at index 1.

- **Linear Indexing:** Access elements using a single index that counts down columns sequentially.

  Example:

  ```
  myVector = [10, 20, 30, 40, 50];
  firstElement = myVector(1); % Accesses the first element (10)
  fourthElement = myVector(4); % Accesses the fourth element (40)
  ```

- **Positional Indexing:** Individual entries can be extracted from a matrix by simply specifying the indices inside round brackets. Several entries can be extracted at once by specifying a matrix, or matrices of indices or use the : operator to extract all entries along a certain dimension. The 'end' statement stands for the last index of a dimension.

```
A = magic(6);
B = A(3,5);          % extract the entry 3 rows down, 5 cols over
C = A([1,2,3],4);    % extract the entries (1,4); (2,4); (3,4)
D = A(4,[1,1,1]);    % extract the entry (4,1) three times
E = A([2,5],[3,1]);  % extract the entries (2,3); (2,1); (5,3); (5,1)
F = A(:,4);          % extract the fourth column
G = A(4,:);          % extract the fourth row
H = A(:);            % extract every entry as a column vector
I = A(end,3);        % extract the entry in the last row, 3rd column
J = A(end-1,end-1);  % extract the entry in the second-to-last row & column

K = A(end-4:end,1);  % extract the last three entries from the first column
L = A(2:end,2:end);  % extract everything except the first row and column
M = A(end:-1:1,:);   % extract everything with the order of the rows reversed
N = diag(A);         % extract the main diagonal of A
O = diag(rot90(A));  % extract the counter diagonal of A
P = diag(A,-2)   % extract the diagonal entries two diagonals left and below the main
```

- **Logical Indexing:** One can also extract entries using a bit pattern, i.e. a matrix of logical values. Only the entries corresponding to true are returned. This can be particularly useful for selecting elements that satisfy some logical criteria such as being larger than a certain value. Logical matrix can be created by relating a numeric matrix to either a scalar value or matrix of the same size via one of the logical operators, < > <= >= == ~= or by a binary function such as *isprime()* or *isfinite()*.

```
B=A>30

 B =
     1    0    0    0    0    0
     0    1    0    0    0    0
     1    0    0    0    0    0
     0    0    1    0    0    0
     0    0    1    0    0    0
```

One can then use this logical matrix to extract elements from A. In the following line, repeat the call to A > 30 but pass the result directly in, without first storing the interim result.

```
B1 = A(A > 30)                % get all elements in A greater than 30
B = A(isprime(A) & (A > 30))  % get all prime elements in A greater than 30
```

```
B1 =
   35
   31
   32
   36
   33
   34
B =
   31
```

We could also achieve the same result using the *find()* function, which returns the indices of all of the non-zero elements in a matrix. While this command is useful when the indices themselves are of interest, using *find()* can be slightly slower than logical indexing although it is a very common code idiom.

```
B2 = A(find(A > 30))          % same result as A(A>30) but calculated differently
```

```
B2 =
   35
```

31

32

36

33

34

One can check that two matrices are equal, (i.e. the same size with the same elements) with the *isequal()* function. Using the == relation returns a matrix of logical values, not a single value.

```
test = isequal(B1,B2)
test2 = all(B1==B2)
```

```
test=
      1
test=
      2
```

Variable Manipulation

- **Assignment Operation:** Manipulation involves modifying existing data or changing how variables are structured. Modify specific elements of an existing array.

```
% myVector is now [10, 20, 30, 40, 50]
myVector(2) = 25;    % Changes the second element from 20 to 25
% myVector is now [10, 25, 30, 40, 50]
% Indexing on the left side of the assignment operator A = [1 2 3];
indices = [1, 3];
A(indices) = [10, 30];     % Assigns 10 to A(1) and 30 to A(3)
% A is now [10, 2, 30]
```

Similarly, anyone change a value or values in a matrix, are performed in a very similar way to the indexing operations above. Both parallel and logical indexing can be used. One indicates which entries will be changed by performing an indexing operation on the left-hand side and then specify the new values on the right-hand side. The right must be either a scalar value, or a matrix with the same dimensions as the resulting indexed matrix on the left. Matlab automatically expands scalar values on the right to the correct size.

```
A=magic(6);
A(3,2) = 999;           % assign 999 to entry (3,2)
A(:,1:3:end) = 999;     % assign 999 to every third column
A(:,1) = [2;3;5;9;8;7]; % assign new values to the first column.
A(A == 999) = 444;      % assign all entries equal to 999 the value 444
```

One can assign every value at once by using the colon operator. The following command temporarily converts A to a column vector, assigns the values on the right-hand side and converts back to the original dimensions.

```
A(:) = 1:36                                                              A =
    1     7    13    19    25    31
    2     8    14    20    26    32
    3     9    15    21    27    33
    4    10    16    22    28    34
    5    11    17    23    29    35
    6    12    18    24    30    36
```

- **Deletion**
  Assigning [] deletes the corresponding entries from the matrix. Only deletions that result in a rectangular matrix are allowed.

```
A([1,3],:) = []        % delete the first and third rows from A
A(:,end) = []          % delete the last column from A
```

- **Expansion**
  When the indices in an assignment operation exceed the size of the matrix, Matlab, rather than giving an error, quietly expands the matrix for you. If necessary, it pads the matrix with zeros. Using this feature is somewhat inefficient, however, as Matlab must reallocate a sufficiently large chunk of contiguous memory and copy the array. It is much faster to preallocate the maximum desired size with the zeros command first, whenever the maximum size is known in advance.

```
[nrows,ncols] = size(A)
A(4,10) = 222
nrows =
        1
ncols =
        4
```

A =
Columns 1 through 7

```
2   1   1   1   0   0   0
0   0   0   0   0   0   0
0   0   0   0   0   0   0
0   0   0   0   0   0   0
```

Columns 8 through 10

```
0   0   0
0   0   0
0   0   0
0   0   222
```

Again, one can extend matrix by concatenating with another matrix.

Method	Horizontal Concatenation	Vertical Concatenation
Bracket Syntax	[A, B] or [A B]	[A; B]
Function	horzcat(A, B)	vertcat(A, B)

- **Reshaping and Replication**
  It is sometimes useful to reshape an array of size m-by-n to size p-by-q where m*n = p*q. The reshape() function lets you do just that. The elements are placed in such a way so as to preserve the order induced by linear indexing. In other words, if a(3) = 3 before reshaping, a(3) will still equal 3 after reshaping.

```
A = zeros(5,6);
A(1:30) = 1:30
B = reshape(A,3,10)
check = A(11) == B(11)
```

```
A =
    1    6   11   16   21   26
    2    7   12   17   22   27
    3    8   13   18   23   28
    4    9   14   19   24   29
    5   10   15   20   25   30
B =
  Columns 1 through 7
    1    4    7   10   13   16   19
    2    5    8   11   14   17   20
    3    6    9   12   15   18   21
```

```
  Columns 8 through 10
    22    25    28
    23    26    29
    24    27    30
check =
     1
```

Further, the repmat() function can be used to tile an array m-by-n times.

```
A = [1 2 ; 3 4]
B = repmat(A,3,6)       % copy A vertically 6 times and horizontally 3 times
```

```
A =
     1     2
     3     4

B =
  Columns 1 through 7
     1     2     1     2     1     2     1
     3     4     3     4     3     4     3
     1     2     1     2     1     2     1
     3     4     3     4     3     4     3
     1     2     1     2     1     2     1
     3     4     3     4     3     4     3
  Columns 8 through 12
     2     1     2     1     2
     4     3     4     3     4
     2     1     2     1     2
     4     3     4     3     4
     2     1     2     1     2
     4     3     4     3     4
```

```
A = 3
A(1:5,1:5) = 3
```

```
A =
     3
A =
     3     3     3     3     3
     3     3     3     3     3
     3     3     3     3     3
     3     3     3     3     3
     3     3     3     3     3
```

Practice Problem

1. (Variable Manipulation): Swap two numbers without using third variable.

2. (Indexing): Reverse a number by using index.

3. The Mirror Test (Palindrome Number)

   A **Palindrome Number** is a number that remains the same when its digits are reversed. It reads identical from left-to-right and right-to-left.
   Example: 12321 is a palindrome because reversing it results in 12321. 123 is not a palindrome because reversing it results in 321.
   Write a MATLAB script to check a numbers is palindrome or not between 50 and 500 using **Auto Initialization** and **string indexing** (the end:-1:1 method)

4. The Logic of Sum and Product (Spy Number)

   A **Spy Number** is a number where the sum of its digits is exactly equal to the product of its digits.
   Example: 1124 is a Spy Number because the sum 1+1+2+4=8 and the product $1\times1\times2\times4=8$ are equal.
   Create a 2×3matrix in MATLAB containing the numbers [123, 22, 5; 10, 1124, 88]. Using **linear indexing** and a for loop, iterate through every element. For each element, convert it to a numeric vector of digits using the **ASCII indexing trick** (num2str(n) - '0') and determine if it is a Spy Number.

5. Power to the Digits (Armstrong Number)

   An **Armstrong Number** (or Narcissistic Number) is a number that equals the sum of its digits, each raised to the power of the total number of digits in that number.
   Example: 153 has 3 digits. So, $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$. Therefore, it is an Armstrong Number.
   Example: 1634 has 4 digits. $1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634$
   Write a MATLAB program that automatically initializes a vector of numbers from 1 to 1000. Use **vectorized operations** (specifically the **dot-power operator** .^) and **logical indexing** to find and display all Armstrong Numbers within this range.

6. Construct a 6 × 6 matrix named `Matrix` with the following specifications: In a 6x6 grid, the blocks will sit perfectly in the corners:

```
        [ C1  C2  C3  C4  C5  C6 ]
[R 1]   [ B   B   .   .   C   C ]
[R 2]   [ B   B   .   .   C   C ]
[R 3]   [ .   .   A   .   .   . ]   (Diagonal element 30)
[R 4]   [ .   .   .   A   .   . ]   (Diagonal element 40)
[R 5]   [ D   D   .   .   E   E ]
[R 6]   [ D   D   .   .   E   E ]
```

Where,

i. Diagonal: Vector `A = [10, 20, 30, 40, 50, 60]`.
ii. Top-Left: Identity matrix `B`
iii. Top-Right: Random matrix `C`.
iv. Bottom-Left: Construct a constant matrix `D` (all elements = 0.88) by using ones matrix
v. Bottom-Right: Sparse matrix `E`

Now perform the following tasks

i. Linear Indexing: Manipulate A times 5, Access the last element of the last row
ii. Positional Indexing: Make a single matrix 4×4 which contains B,C,D,E maintaining same position.
iii. Logical Indexing: Find all values greater than 5 and replace them with 100, zero out all elements that are exactly 0.88.
iv. Manipulation: Swap the 1st row with the 6th row, Square all elements in the center area (rows 3-4, cols 3-4)

Experiment 6: Importing data, reading and writing excel files and learning other miscellaneous features

Objectives

After completing this experiment, students will be able to:

- Importing Excel data into MATLAB
- Reading and writing data to Excel files
- Performing basic data processing and analysis
- Using commonly used miscellaneous MATLAB commands

Introduction

In modern industrial and production engineering applications, engineers frequently work with large datasets related to production output, machine utilization, quality control, inventory, and energy consumption etc. These datasets are often stored in spreadsheet formats such as Microsoft Excel. MATLAB provides powerful tools to import, analyze, process, and export such data efficiently.

So, this experiment focuses on developing structured MATLAB programs by introducing students to import data, read and write excel file and other miscellaneous features. This experiment moves toward program design and logical flow control. Students will learn how to:

i. Import Excel data using different MATLAB functions
ii. Read numerical and text data from Excel files
iii. Filter the raw data
iv. Write processed data and results back to Excel
v. Use basic MATLAB commands for data manipulation
vi. Generate simple statistical summaries and plots
vii. Prepare a professional project report

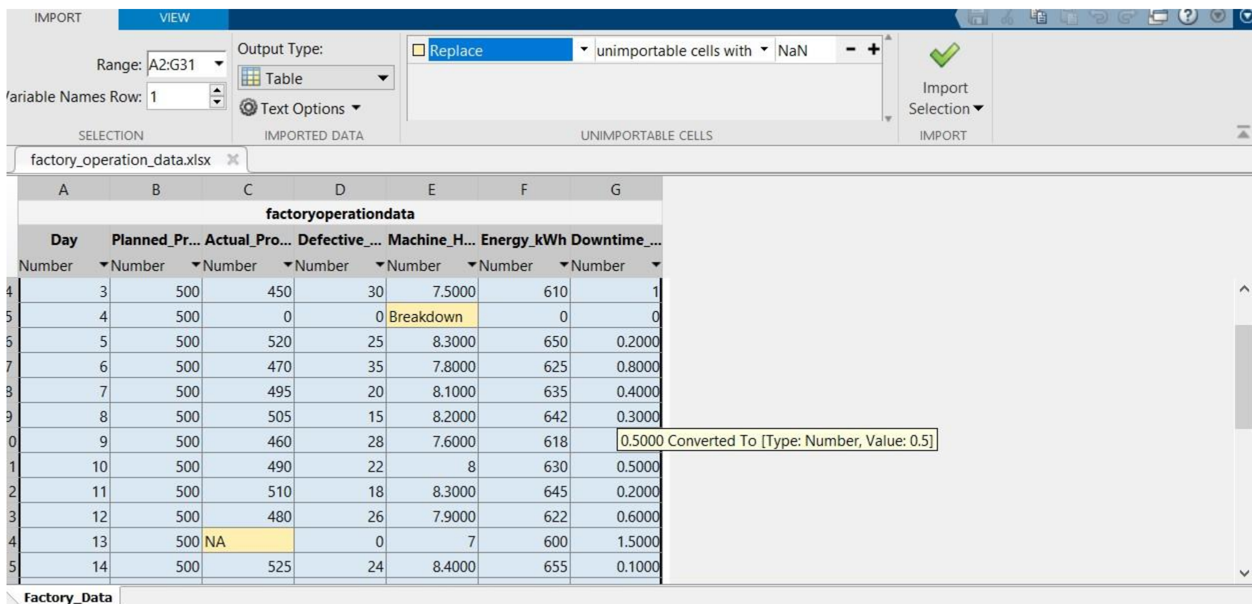
Importing Data from Excel (Read Data)

a) Interactive Method (Import Tool)

This is a good method for one-off imports or to generate code for repeated operations.

1. Place your Excel file in your current MATLAB folder.
2. Double-click the file name in the Current Folder panel, or click the Import Data icon on the Home tab.

3. The Import Tool window opens, showing a preview of the data.
4. Select the data range you want to import using your mouse.
5. Choose the Output Type (e.g., Table, Numeric Matrix, Cell Array, Column vectors).
6. Click Import Selection to bring the data into your workspace.



b) Programmatic Method (Functions)

For use within scripts and functions, use the following commands:

- readtable('filename.xlsx'): - Reads data into a **table** array, ideal for mixed data types (numbers and text) and handling headers automatically.
- Readtimetable('filename.xlsx'):- Import tabular data from a spreadsheet into a timetable using the readtimetable function. If the rows of spreadsheet are associated with times, one can import the data as a timetable.
- readmatrix('filename.xlsx'): - Reads data into a **numeric matrix** (all data must be of the same type).
- readcell('filename.xlsx'): - Reads data into a **cell array**, suitable for mixed data when a table isn't preferred.

Example:

```
% Read the entire file as a table
data_table = readtable('my_experiment_data.xlsx');

% Read a specific range as a matrix
numeric_data = readmatrix('my_experiment_data.xlsx', 'Sheet', 'Sheet1',
'Range', 'B2:D4');
```

Note: xlsread('filename.xlsx'): - *The older xlsread function is not recommended for new code, but still works for backward compatibility.*

```
num=xlsread('data.xlsx','Sheet1','A1:B10');
```

detectImportOptions:- The function works by scanning your file and returning a SpreadsheetImportOptions object for Excel.

```
% Detect existing settings in the file
opts = detectImportOptions('myData.xlsx');

% Customize specific properties
opts.Sheet = 'ScientificData';      % Select a specific sheet
opts.SelectedVariableNames = {'Temp', 'Pressure'}; % Pick only certain
columns
opts.DataRange = 'A2';              % Start reading data from cell A2

% Read the table using these options
T = readtable('myData.xlsx', opts);
```

- uiimport -pastespecial:- First copy the data from excell and then type 'uiimport - pastespecial' command in the command window and follow the interactive method.

- importdata:- A quick way to grab numeric or text data from the clipboard.

```
A = importdata('-pastespecial');
```

- clipboard:- Retrieves the raw text content of the clipboard as a string.

```
str = clipboard('paste');
```

Handling Missing Data

- **fillmising:** The fillmissing function remains a primary tool for cleaning data imported from Excel. It replaces standard missing values (like NaN in numeric data or <missing> in strings) with estimated or constant values.

Syntax Type	MATLAB Code Example	Description
Constant	`fillmissing(A, 'constant', v)`	Fills all missing entries with a scalar value `v` (e.g., `0`).

Neighbor Fill	`fillmissing(A, 'previous')`
	
	`fillmissing(A, 'next')`
		Replaces gaps with the last known non-missing value.
		
		Fills with the next available non-missing value.

Interpolation	`fillmissing(A, 'linear')`	Estimates gaps using a straight line between neighbors.
Moving Mean	`fillmissing(A, 'movmean', 5)`	Fills gaps using a local average of 5 neighboring points.
Moving Median	`fillmissing(A, 'movmedian', 10)`	Best for data with outliers; uses the middle value of 10 neighbors.
Table Columns	`fillmissing(T, 'linear', 'DataVariables', {'Var1'})`	Only applies the fill method to specific named columns in a table.
Mixed Constants	`fillmissing(T, 'constant', {0, 'N/A'})`	Uses different fill values for numeric vs. text columns in a table.

- **ismissing:** The ismissing function returns a logical array (0s and 1s) of the same size as your input, indicating where data is absent. One can specify non-standard missing values (e.g., -99 or "N/A") frequently found in industrial Excel logs.

```
% Find standard missing values and custom -99 indicators
mask = ismissing(myData, [-99, "N/A"]);
```

- **any:** The any function determines if any element in a vector or matrix is non-zero (true). When paired with ismissing, it helps you find entire rows or columns that contain at least one error.

Locate all missing data: mask = ismissing(myData);

Find Rows with Missing Data: Use any(mask, 2) to check across rows.

Find Columns with Missing Data: Use any(mask, 1) to check down

columns. Find any data is missing: Use anymissing(myData) to check any

missing value

- **rmmissing:** Simply removes any rows with missing values (NaN) instead of filling them.
- **standardizeMissing:** Converts non-standard placeholders in Excel (like 999 or "Unknown") into official MATLAB missing types (NaN or <missing>).
- **unique:** Removes duplicate rows or values from a dataset.
- **normalize:** Scales data to a common range (e.g., Z-score or 0-to-1) to ensure variables are comparable.

Exporting Data to Excel (Write Data)

Similar to reading, specific functions are used based on the MATLAB data type you are exporting.

- writetable(T, 'filename.xlsx'):- Writes a **table** to an Excel file.
- writematrix(M, 'filename.xlsx'):- Writes a **matrix** to an Excel file.
- writecell(C, 'filename.xlsx'):- Writes a **cell array** to an Excel file.

Note: xlswrite('filename.xlsx'): - *The older xlswrite function is not recommended for new code, but still works for backward compatibility.*

```
% Create sample data
A = rand(5, 4); % A 5x4 matrix of random numbers
T = array2table(A, 'VariableNames', {'Col1', 'Col2', 'Col3', 'Col4'}); %
Convert to table

% Write the table to a new file
writetable(T, 'output_data.xlsx');

% Write a matrix to a specific sheet and range
writematrix(A, 'output_data.xlsx', 'Sheet', 'Sheet2', 'Range', 'C1');
```

Practice Problem

A manufacturing plant records daily production (30 days) information in an Excel file. The dataset is shown below:

Day	Planned Production	Actual Production	Defective Units	Machine Hours	Energy kW/h	Downtime hr.
1	500	480	18	8	620	0.5
2	500	510	22	8.2	640	0.3
3	500	450	30	7.5	610	1
4	500	0	0	Breakdown	0	0
5	500	520	25	8.3	650	0.2
6	500	470	35	7.8	625	0.8
7	500	495	20	8.1	635	0.4
8	500	505	15	8.2	642	0.3
9	500	460	28	7.6	618	0.9
10	500	490	22	8	630	0.5
11	500	510	18	8.3	645	0.2
12	500	480	26	7.9	622	0.6
13	500	NA	0	7	600	1.5
14	500	525	24	8.4	655	0.1
15	500	500	21	8.2	640	0.3
16	500	470	29	7.7	620	0.8
17	500	495	23	8.1	635	0.4
18	500	510	19	8.3	648	0.2
19	500	485	27	7.9	628	0.6
20	500	0	0	0	Not Given	0
21	500	520	20	8.3	650	0.2
22	500	505	18	8.2	642	0.3
23	500	460	32	7.6	618	0.9
24	500	490	24	8	630	0.5
25	500	515	19	8.3	646	0.2
26	500	480	28	7.8	623	0.7
27	500	500	22	8.1	638	0.4
28	500	510	17	8.3	645	0.2
29	500	470	30	7.7	620	0.8
30	500	495	21	8.1	635	0.4

Tasks:

1. Import and read the data in MATLAB by interactive method/ programmatic method
2. Filter the data e.g. Remove irrelevant data / Fill the 'NaN' values
3. Calculate actual factory KPIs: production efficiency, defect rate, downtime, energy per unit
4. Visualize data
5. Writes final results back to Excel for the report

Experiment 7(a): Solving Equations of Linear Algebra Using Gauss Elimination Method

Objectives

After completing this experiment, students will be able to:

- To implement the Gauss Elimination method using MATLAB programming.
- To apply forward elimination and backward substitution in MATLAB to solve systems of linear equations.
- To develop proficiency in MATLAB matrix operations and loop-based numerical computation.

Introduction

The Gauss Elimination method is a fundamental numerical technique used to solve systems of simultaneous linear equations. The method systematically reduces the given system into an equivalent form that is easier to solve.

The procedure involves two main stages:
Stage-1: Forward Elimination, where the augmented matrix is transformed into an upper triangular form.
Stage-2: Backward Substitution, where the unknown variables are calculated starting from the last equation.

Working Rule

Consider the system of equations

$$a_{11}x + a_{12}y + a_{13}z = b_1$$
$$a_{21}x + a_{22}y + a_{23}z = b_2$$
$$a_{31}x + a_{32}y + a_{33}z = b_3$$

In matrix form

$$AX = B$$
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Augmented matrix

$$C = [A{:}\,B] = \begin{bmatrix} a_{11} & a_{12} & a_{13}{:}\,b_1 \\ a_{21} & a_{22} & a_{23}{:}\,b_2 \\ a_{31} & a_{32} & a_{33}{:}\,b_3 \end{bmatrix}$$

Reduce the augmented matrix to echelon form using elementary row transformations,

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13}: d_1 \\ 0 & c_{22} & c_{23}: d_2 \\ 0 & 0 & c_{33}: d_3 \end{bmatrix}$$

After Gauss elimination, the system becomes upper triangular:

$$c_{11}x + c_{12}y + c_{13}z = d_1$$
$$c_{22}y + c_{23}z = d_2$$
$$c_{33}z = d_3$$

Start from the last equation (only one unknown): $z = \frac{d_3}{c_{33}}$

Substitute $z$into the second equation: $y = \frac{d_2 - c_{23}z}{c_{22}}$

Substitute $y$and $z$into the first equation:$x = \frac{d_1 - c_{12}y - c_{13}z}{c_{11}}$

MATLAB Implementation

Problem: Gauss Elimination to Find an Augmented Upper Triangular Matrix.
Given the system of linear equations:

$$\begin{cases} 6x_1 + 1x_2 - 6x_3 - 5x_4 = 6 \\ 4x_1 - 3x_2 + 0x_3 + 1x_4 = -7 \\ 2x_1 + 2x_2 + 3x_3 + 2x_4 = -2 \\ 0x_1 + 2x_2 + 0x_3 + 1x_4 = 0 \end{cases}$$

Task: Use the Gauss Elimination method to convert the augmented matrix of the system into upper triangular form.

Coefficient matrix A:

$$A = \begin{bmatrix} 6 & 1 & -6 & -5 \\ 4 & -3 & 0 & 1 \\ 2 & 2 & 3 & 2 \\ 0 & 2 & 0 & 1 \end{bmatrix}$$

RHS vector C:

$$C = \begin{bmatrix} 6 \\ -7 \\ -2 \\ 0 \end{bmatrix}$$

Augmented matrix [A|C] combines them into **one matrix**:

$$\text{Aug} = \begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 4 & -3 & 0 & 1 & -7 \\ 2 & 2 & 3 & 2 & -2 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix}$$

```
% Gauss Elimination Method

% Step 1: Define the coefficient matrix (A) and RHS vector (C)
A = [6 1 -6 -5; 4 -3 0 1; 2 2 3 2; 0 2 0 1];    % Coefficient matrix
C = [6; -7; -2; 0];                             % Right-hand side vector

% Step 2: Check if the matrix is square
[m, n] = size(A);     % Get number of rows (m) and columns (n)
if m ~= n
    dis('Matrix must be square to apply Gauss Elimination');
end

% Step 3: Form the augmented matrix [A | C]
Aug = [A C];
disp('Initial Augmented Matrix [A | C]:');
disp(Aug);

% Step 4: Forward Elimination
for k = 1: n-1            % k selects the pivot position
                          % Pivot element = Aug(k,k)
    for i = k+1: n           % i selects rows below the pivot row
                             % These rows will be modified to make zeros
        factor = Aug(i,k) / Aug(k,k);
        for j = k: n+1    % j selects columns to be updated
                          % Start from pivot column and include RHS (n+1)
            Aug(i,j) = Aug(i,j) - factor * Aug(k,j);
        end
    end
end
disp('Upper Triangular Augmented Matrix:');
disp(Aug);
```

Explanation

for k = 1: n-1

k represents the pivot step

k = 1 ⟶ eliminate terms below a11

k = 2 ⟶ eliminate terms below a22

…

k = n−1 ⟶ eliminate terms below a (n−1, n−1)

No work is needed for k = n, because nothing is below the last row

for i = k+1: n
i represents the row being updated in the current pivot step
i = k+1 → first row below the pivot
i = k+2, k+3 … n → remaining rows below the pivot
Pivot row k and rows above are not changed

for j = k: n+1
j represents the column being updated in the current row i
j = k → pivot column, to eliminate the element
j = k+1, k+2 … n+1 → remaining matrix entries that must be updated

```
% Step 5: Backward Substitution
x = zeros(n,1);                 % Initialize solution vector
x(n) = Aug(n, n+1) / Aug(n,n);  % Solve the last variable directly

for i = n-1: -1: 1
    sum = 0;                    % Reset summation for each equation
    for j = i+1: n              % Add known variable contributions
        sum = sum + Aug(i,j) * x(j);
    end
    x(i) = (Aug(i,n+1) - sum) / Aug(i,i);   % Compute the current variable
end

% Step 6: Display the solution
disp('Solution of the system [x1, x2, ..., xn]:');
disp(x);
```

Explanation

for i = n-1: -1: 1
selects the current row (variable) to solve.

- i = n-1 → solve second-to-last variable x(n-1)
- i = n-2 → solve third-to-last variable x(n-2)
- …
- i = 1 → solve first variable x(1)

for j = i+1: n
sums contributions from already known variables in row i.

- j = i+1 → first known variable to the right of pivot in row i
- j = i+2, i+3 … n → remaining known variables in the row

Exercises

1. Solve the following system of linear equations using the Gauss Elimination method:

$$\begin{aligned} 2x_1 + 2x_2 + x_3 &= 6 \\ 4x_1 + 2x_2 + 3x_3 &= 4 \\ x_1 - x_2 + x_3 &= 0 \end{aligned}$$

2. Solve the following system of linear equations using the Gauss Elimination method:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 10 \\ 2x_1 + 3x_2 + x_3 + 2x_4 &= 20 \\ x_1 + 2x_2 + 3x_3 + x_4 &= 16 \\ 3x_1 + x_2 + 2x_3 + 2x_4 &= 19 \end{aligned}$$

3. Solve the following system of linear equations using the Gauss Elimination method:

$$\begin{aligned} 4x_1 - x_2 + 2x_3 &= 1 \\ 3x_1 + 2x_2 - x_3 + x_4 &= 4 \\ 2x_1 - 3x_2 + x_3 + 2x_4 &= -2 \\ x_1 + x_2 + x_3 + x_4 &= 6 \end{aligned}$$

Experiment 7(b): Solving Equations of Linear Algebra Using Jacobi Iteration & Gauss-Seidel Iteration Method

Objectives

After completing this experiment, students will be able to:

- Implement the Jacobi and Gauss–Seidel methods in MATLAB to solve systems of linear equations iteratively.
- Observe and analyze the convergence behavior of these iterative methods using MATLAB outputs.
- Assess the efficiency and reliability of both methods by comparing iteration counts and accuracy in MATLAB simulations

Introduction

Iterative methods are widely used for solving systems of linear equations, especially when the system size is large or the coefficient matrix is sparse, conditions under which direct solution methods often suffer from increased round-off errors and high memory requirements. This experiment focuses on the implementation of the Jacobi and Gauss–Seidel iterative methods, which obtain approximate solutions by repeatedly updating the unknown variables using values from previous iterations.

1. Jacobi Iteration Method

Given a system of linear equations:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

From equation

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n)$$

$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n)$$

$$x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})$$

General form for the i-th equation:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\, x_j \right), i = 1,2,\ldots,n$$

Then make an initial guess of the solution $\mathbf{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\right)$**.** Substitute these values into the right-hand side of the rewritten equations to obtain the *first approximation,* $\mathbf{x}^{(1)} = \left(x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}\right)$.

This accomplishes one **iteration**.

In the same way, the *second approximation,* $\mathbf{x}^{(2)} = (x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)})$ is computed by substituting the first approximation's values into the right-hand side of the rewritten equations.

By repeated iterations, we form a sequence of approximations

Iterate: For $k = 1{,}2{,}3, \ldots$, compute:

$$\mathbf{x}^{(k)} = (x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)})$$

General formula for $k \geq 1$, $i = 1, \ldots, n$:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\, x_j^{(k-1)} \right]$$

MATLAB Implementation

Use the Jacobi iterative method to solve the system of linear equations:

$$\begin{cases} 4x_1 + 1x_2 + 2x_3 = 16 \\ 1x_1 + 3x_2 + 1x_3 = 10 \\ 1x_1 + 2x_2 + 5x_3 = 12 \end{cases}$$

Given the initial guess $\mathbf{x}^{(0)} = \mathbf{0}$, stopping tolerance 10^{-6}, and a maximum of 1000 iterations, find the approximate solution vector $\mathbf{x}$

MATLAB Implementation

```
% Define the system A*x = b
A = [4 1 2; 1 3 1; 1 2 5];                  % Coefficient matrix
b = [16; 10; 12];                           % Right-hand side vector

tol = 0.000001;                             % Stopping tolerance for convergence
maxIter = 1000;                             % Maximum number of iterations allowed

% Get matrix size and check if A is square
[m,n] = size(A);                       % m = number of rows, n = number of columns
if m ~= n
    error('Matrix A must be square');
end

% Check if matrix A is strictly diagonally dominant
for i = 1:n
    diagonal = abs(A(i,i));        % Absolute value of diagonal element a_ii
      offDiagonalSum = sum(abs(A(i,:))) - diagonal;% Sum of absolute values
    if diagonal <= offDiagonalSum
        error('Matrix A must be diagonally dominant');
    end
end

x_old = zeros(n,1);            % Initial guess for solution (previous iteration)
x_new = zeros(n,1);            % Storage for new solution (current iteration)

% Jacobi Iteration Loop
for k = 1:maxIter
    for i = 1:n
        sumVal = 0;                % Initialize sum of a_ij * x_j for j ≠ i
        for j = 1:n
            if j ~= i
                sumVal = sumVal + A(i,j) * x_old(j);
            end
        end
        x_new(i) = (b(i) - sumVal) / A(i,i);
    end

% Check for convergence
  iterationChange = max(abs(x_new - x_old));    % Maximum absolute difference
    if iterationChange <= tol
             break;
```

```
    end

% Update old solution for next iteration
    x_old = x_new;
end

% Display final result
disp('Solution vector x (Jacobi Method):');
disp(x_new);
disp(k);
```

2. Gauss-Seidel Iteration Method

Given a system of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

From equation

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n)$$

$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n)$$

$$x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})$$

General form for the i-th equation:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\, x_j\right), i = 1,2,\ldots,n$$

Start with an initial guess of the solution:

$$\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})$$

Substitute these values into the right-hand side of the rewritten equations sequentially, using the most recently updated values, to obtain the first approximation.

First iteration:

$$x_1^{(1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - \cdots - a_{1n}x_n^{(0)})$$

$$x_2^{(1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(0)} - \cdots - a_{2n}x_n^{(0)})$$

$$x_3^{(1)} = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)} - a_{34}x_4^{(0)} - \cdots - a_{3n}x_n^{(0)})$$

$$\ldots$$

$$x_n^{(1)} = \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(1)} - a_{n2}x_2^{(1)} - \cdots - a_{n,n-1}x_{n-1}^{(1)})$$

This gives the first approximation:

$$\mathbf{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)})$$

Second iteration:

$$x_1^{(2)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(1)} - a_{13}x_3^{(1)} - \cdots - a_{1n}x_n^{(1)})$$

$$x_2^{(2)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(2)} - a_{23}x_3^{(1)} - \cdots - a_{2n}x_n^{(1)})$$

$$x_3^{(2)} = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(2)} - a_{32}x_2^{(2)} - a_{34}x_4^{(1)} - \cdots - a_{3n}x_n^{(1)})$$

$$\ldots$$

$$x_n^{(2)} = \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(2)} - a_{n2}x_2^{(2)} - \cdots - a_{n,n-1}x_{n-1}^{(2)})$$

This gives the second approximation:

$$\mathbf{x}^{(2)} = (x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)})$$

By repeated iterations,

$$\mathbf{x}^{(k)} = (x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), k = 1,2,3, \ldots$$

General formula for $k \geq 1, i = 1, \ldots, n$:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k-1)}\right]$$

With the Jacobi method, the values of $x_i^{(k)}$ obtained in the kThe iteration remains unchanged until the entire $(k+1)$The iteration has been calculated. With the Gauss-Seidel method, the new values $x_i^{(k+1)}$ is used, as soon as the values are known.

MATLAB Implementation

Use the Gauss-Seidel iterative method to solve the system of linear equations:

$$\begin{cases} 4x_1 + 1x_2 + 2x_3 = 16 \\ 1x_1 + 3x_2 + 1x_3 = 10 \\ 1x_1 + 2x_2 + 5x_3 = 12 \end{cases}$$

Given the initial guess $\mathbf{x}^{(0)} = \mathbf{0}$, stopping tolerance 10^{-6}, and a maximum of 1000 iterations, find the approximate solution vector **x.**

MATLAB Implementation

```
% Define the system A*x = b
A = [4 1 2; 1 3 1; 1 2 5];          % Coefficient matrix
b = [16; 10; 12];                   % Right-hand side vector

% Set tolerance and maximum number of iterations
tol = 0.000001;           % Stopping tolerance for convergence (change < tol)
maxIter = 1000;           % Maximum number of iterations allowed

% Get matrix size and check if A is square
[m,n] = size(A);          % m = number of rows, n = number of columns
if m ~= n
    error('Matrix A must be square');
end

% Check if matrix A is strictly diagonally dominant
for i = 1:n
    diagonal = abs(A(i,i));         % Absolute value of diagonal element a_ii
    offDiagonalSum = sum(abs(A(i,:))) - diagonal;   % Sum of absolute values
if diagonal <= offDiagonalSum
        error('Matrix A must be diagonally dominant');
    end
end

% Initial guess for the solution
x = zeros(n,1);           % Initial guess is chosen as zero vector

% Gauss-Seidel Iteration Process
for k = 1:maxIter
     x_old = x;           % store previous iteration values

    for i = 1:n
        sumVal = 0;

               for j = 1:i-1                  % for j < I, updated values
            sumVal = sumVal + A(i,j) * x(j);
        end

                for j = i+1:n                    % for j > I, old values
            sumVal = sumVal + A(i,j) * x_old(j);
        end
```

```
        x(i) = (b(i) - sumVal) / A(i,i);
    end

 % Check for convergence
    iterationChange = max(abs(x - x_old));

    if iterationChange <= tol
                break;
    end
end

% Display final solution
disp('Solution vector x (Gauss-Seidel Method):');
disp(x);
disp(k);
```

Exercises

1. Use the Jacobi/Gauss-Seidel method to solve the system of linear equations:

$$\begin{cases} 5x_1 + x_2 + x_3 = 7 \\ x_1 + 4x_2 + x_3 = 6 \\ x_1 + x_2 + 3x_3 = 5 \end{cases}$$

Given the initial guess $\mathrm{x}^{(0)} = 0$, stopping tolerance 10^{-6}, and a maximum of 1000 iterations, find the approximate solution vector x.

2. Use the Jacobi/Gauss-Seidel method to solve the system of linear equations:

$$\begin{cases} 10x_1 - x_2 + 2x_3 = 6 \\ -x_1 + 8x_2 - x_3 = 5 \\ 2x_1 - x_2 + 7x_3 = 9 \end{cases}$$

Given the initial guess $\mathrm{x}^{(0)} = 0$, stopping tolerance 10^{-6}, and a maximum of 1000 iterations, find the approximate solution vector x.

Experiment 7(c): Solving ODEs using MATLAB

Objectives

Upon completion of this experiment, students will be able to:

- Classify and interpret different types of Ordinary Differential Equations (ODEs).
- Implement Euler's method and Runge–Kutta methods in MATLAB.
- Solve initial value problems using MATLAB's built-in solver ode45.
- Convert higher-order ODEs into systems of first-order equations.
- Compare fixed-step numerical methods with adaptive solvers.

Introduction

An Ordinary Differential Equation (ODE) is an equation involving a function and its derivatives with respect to one independent variable. ODEs model dynamic systems like population growth, mechanical vibrations, or chemical reactions.

- **Order of an ODE:** The order of an ODE is determined by the highest derivative present.

$$\frac{dy}{dt} = -2y \text{ (First order)}$$

$$\frac{d^2y}{dt^2} + 3\frac{dy}{dt} + 2y = 0 \text{ (Second order)}$$

Linear vs. Nonlinear: An ODE is linear if the dependent variable and its derivatives appear only to the first power and are not multiplied together.
Linear example:

$$\frac{dy}{dt} + 3y = t$$

Nonlinear example:

$$\frac{dy}{dt} = y^2$$

Initial Value Problem (IVP): An IVP consists of:

$$\frac{dy}{dt} = f(t, y), y(t_0) = y_0$$

The objective is to determine $y(t)$ over a given interval.

Analytical Methods for Solving ODEs

Analytical solutions (exact, closed-form) work for simple cases but fail for complex or nonlinear ODEs. For a first order separable ODE, $\frac{dy}{dt} = f(t)g(y)$; the general form of solution is obtained by integrating both sides, which is $\int \frac{dy}{g(y)} = \int f(t)\, dt$.

Example: $\frac{dy}{dt} = ky$

Solution: $y(t) = y_0 e^{kt}$.

Analytical methods are limited to specific forms. In practical engineering systems, analytical solutions are often unavailable. For real-world problems (e.g., nonlinear or variable coefficients), we use numerical methods.

Numerical Methods of Solving ODE:

The simplest numerical method of solving an ODE is Euler's method. Euler's method uses one slope:

$$y_{n+1} = y_n + hf(t_n, y_n);\ \text{where h is the step size}$$

Euler's method uses slope only at the start. It results in poor accuracy, and error accumulates quickly. For a better estimate, we use Runge-Kutta (RK) method. A Runge–Kutta method is a one-step numerical method of the form:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i\, k_i$$

where the stage slopes k_i are defined by:

$$k_i = f\left(t_n + c_i h,\ y_n + h \sum_{j=1}^{i-1} a_{ij}\, k_j \right)$$

Here:

- s= number of stages; a_{ij}, b_i, c_i are constants defining the method

The core idea of RK method is instead of using one slope, we sample slopes at multiple points, combine them intelligently to get a better estimate of the average slope. RK methods estimate the solution by:

- Sampling the slope at multiple points
- Taking a weighted average of slopes

Second-order Runge–Kutta (Heun's Method)

A common second-order RK method (Heun's method) is defined as:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h,\ y_n + hk_1)$$
$$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2)$$

In case of RK2, we take a slope at the beginning as well as the end of the interval. The resulting slope is the average of the two slopes. It reduces truncation error.

Fourth-Order Runge–Kutta (RK4)

The classical 4th-order Runge–Kutta method is defined by:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_1\right)$$
$$k_3 = f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_2\right)$$
$$k_4 = f(t_n + h,\ y_n + hk_3)$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

RK4 achieves high accuracy. It is stable and only requires first derivatives. The coefficients are chosen so that the Taylor series expansion of the numerical method matches the true solution up to terms of order h^4.

MATLAB Implementation (manually)

RK2 (Heun's Method):

$$\frac{dy}{dt} = -2y;\ y(0) = 1;\ \text{h} = 0.1;\ \text{timespan} = [0,1]$$

```
% Step 1: Define the differential equation
f = @(t, y) -2*y;


% Step 2: Define step size and time interval
h = 0.1;
t_start = 0;
t_end = 1;
t = t_start:h:t_end;


% Step 3: Prepare storage for solution
y = zeros(1, length(t));

y(1) = 1;% Initial condition y(0) = 1


% Step 4: Apply Heun's Method (RK2)
for i = 1:length(t)-1

    k1 = f(t(i), y(i)); % Slope at the beginning of the interval


    k2 = f(t(i) + h, y(i) + h*k1); % Slope at the end of the interval

    y(i+1) = y(i) + (h/2)*(k1 + k2); % Average the two slopes

end

% Step 5: Plot the result
plot(t, y, 'o-')
grid on
xlabel('Time')
ylabel('y(t)')
title('Heun Method (RK2)')
```

RK4 Method:

$$\frac{dy}{dt} = y - t^2 + 1;\ \ y(0) = 0.5;$$
$$h = 0.2;\ \text{timespan} = [0,2]$$

```
% Define differential equation
f = @(t,y) y - t.^2 + 1;

% Numerical parameters
h = 0.2;
t = 0:h:2;
n = length(t);

% Preallocate solution vector
y = zeros(size(t));
y(1) = 0.5;    % Initial condition


% RK4 loop
for i = 1:n-1

    k1 = f(t(i), y(i));
    k2 = f(t(i) + h/2, y(i) + (h/2)*k1);
    k3 = f(t(i) + h/2, y(i) + (h/2)*k2);
    k4 = f(t(i) + h, y(i) + h*k3);

    y(i+1) = y(i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);

end

% Plot
plot(t, y, 'o-')
grid on
xlabel('Time')
ylabel('y(t)')
title('RK4 Solution')
```

MATLAB ODE Solvers

MATLAB offers a suite of built-in ODE solvers designed to handle various initial value problems by numerically integrating systems of first-order differential equations. The most common general-purpose solver is **ode45**, which utilizes an explicit Runge-Kutta (4,5) formula to adaptively adjust its step size for optimal accuracy and efficiency. While ode45 is the recommended first choice for most non-stiff problems, MATLAB provides specialized alternatives for different scenarios: ode15s is better suited for "stiff" systems where timescales vary significantly, and ode113 is often more efficient for problems requiring high precision or involving computationally expensive function evaluations. In this lecture, we will focus on ode45.

ode45 numerically solves initial value problems of the form:

$$\frac{dy}{dt} = f(t, y), y(t_0) = y_0$$

Internally, it uses Runge–Kutta 4th/5th order, adaptive step size and automatic error control. Unlike manual method, user does not choose step size. It is determined by MATLAB.

General syntax: `[t, y] = ode45(odefun, tspan, y0)`

Input 1: odefun

This defines the differential equation.

Mathematical meaning

$$\frac{dy}{dt} = f(t, y)$$

MATLAB requirement: It must be a function of two (independent and dependent) variables.

```
dydt = odefun(independent_variable, dependent_variable)
```

Example: `f = @(t, y) -2*y;`

This corresponds to:

$$\frac{dy}{dt} = -2y$$

Input 2: tspan (time interval)

```
tspan = [t0 tf];
```

Example: `tspan = [0 10];` [Start simulation at t = 0, end simulation at t = 10]

It is to be noted that MATLAB chooses internal step sizes (h in previous examples). Hence, output t values are not equally spaced.

Input 3: y0 (initial condition)

Example (scalar ODE): `y0 = 5;`

Example (system of ODEs): `y0 = [1; 0];`

Note: A column vector must be used for systems of ODEs.

Outputs: t and y

`[t, y]`

- **t**: Column vector of time points (independent variable)
- **y**: Matrix where each row is the solution at time t(i), each column a state variable.

Solving First-Order ODEs with ode45

$$\frac{dy}{dt} = -2y, y(0) = 5$$

timespan = [0, 5]

MATLAB Code

```
f = @(t, y) -2*y;              % Define the ODE dy/dt = -2y

tspan = [0 5];                 % Time interval
y0 = 5;                        % Initial condition


[t, y] = ode45(f, tspan, y0);  % Solve the ODE


plot(t, y, 'LineWidth', 2)
grid on
xlabel('Time')
ylabel('y(t)')
title('Solution using ode45')
```

Solving Higher Order Derivatives with ode45

MATLAB cannot directly solve higher-order ODEs, the user must convert them to a system of first order ODEs. Numerically, MATLAB only integrates first derivatives. Therefore, all higher derivatives must be represented as additional state variables. Mathematically, ode45 expects problems of this exact form:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

Where, $\mathbf{y}$ is the state vector, the highest derivative is first order. It has no mechanism to directly handle y'', y'''or higher derivatives. So, the problem must be rewritten to fit what the solver understands. Instead of viewing the problem as a function $y(t)$ with higher derivatives, it is dissolved into multiple variables, each with a first derivative. This is called the state-space viewpoint.

For an n-th order ODE:

$$y^{(n)} = F(t, y, y', y'', \dots, y^{(n-1)})$$

We define new variables:

$$\begin{gathered} x_1 = y \\ x_2 = y' \\ x_3 = y'' \\ \vdots \\ x_n = y^{(n-1)} \end{gathered}$$

Then their derivatives are:

$$\begin{gathered} x_1' = x_2 \\ x_2' = x_3 \\ \vdots \\ x_n' = F(t, x_1, x_2, \dots, x_n) \end{gathered}$$

Now every equation is in first order. ode45 can be used to solve it.

Step by step second-order example:

Original ODE

$$y'' + 3y' + 2y = 0; \text{ timespan} = [0, 10]$$

This is **second order**, hence ode45 cannot be used directly.

Step 1: Solve for the highest derivative

$$y'' = -3y' - 2y$$

Step 2: Define state variables

$$x_1 = y$$
$$x_2 = y'$$

Step 3: Write equations for derivatives

$$x_1' = y' = x_2$$
$$x_2' = y'' = -3x_2 - 2x_1$$

The original ODE has resulted into **two first-order equations**.

Step 4: Vector (state-space) form

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -3x_2 - 2x_1 \end{bmatrix}$$

Now it is ready to be provided as input for ode45

MATLAB Implementation

```
f = @(t, X) [
     X(2);                  % x1' = x2
   -3*X(2) - 2*X(1)         % x2' = -3x2 - 2x1
];
X0 = [1; 0];                % Initial values

[t, X] = ode45(f, [0 10], X0);


plot(t, X(:,1))
```

Explanation

```
f = @(t, X) [
    X(2);                  % x1' = x2
   -3*X(2) - 2*X(1)        % x2' = -3x2 - 2x1
];
```

- X(1) = $x_1 = y$
- X(2) = $x_2 = y'$
- Output is a column vector of derivatives

Initial conditions:

Original conditions:

$$y(0) = 1, y'(0) = 0$$

State-space initial vector:

```
X0 = [1; 0];
```

Each state variable needs its own initial value.

Solve using ode45

```
[t, X] = ode45(f, [0 10], X0);
```

What MATLAB now stores:

- X(:,1) → $y(t)$
- X(:,2) → $y'(t)$

Plotting:

```
plot(t, X(:,1))
```

3rd-order example

Original equation

$$\frac{d^3y}{dt^3} + 4\frac{d^2y}{dt^2} + 5\frac{dy}{dt} + 2y = 0$$

timespan = [0, 10]

Rearrange to isolate the highest derivative:

$$\frac{d^3y}{dt^3} = -4\frac{d^2y}{dt^2} - 5\frac{dy}{dt} - 2y$$

Define states

$$\begin{aligned} x_1 &= y \\ x_2 &= \dot{y} \\ x_3 &= \ddot{y} \end{aligned}$$

Write the first-order system

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = x_3$$
$$\dot{x}_3 = -4x_3 - 5x_2 - 2x_1$$

MATLAB implementation

```
odefun = @(t, x) [
    x(2);                      % x1' = y'
    x(3);                      % x2' = y''
    -4*x(3) - 5*x(2) - 2*x(1) % x3' = y'''
];
tspan = [0 10];

x0 = [1; 0; 0];

[t, x] = ode45(odefun, tspan, x0);

plot(t, x(:,1), 'LineWidth', 2)
grid on
xlabel('Time')
ylabel('y(t)')
```

Explanation:

```
odefun = @(t, x) [
    x(2);                    % x1' = y'
    x(3);                    % x2' = y''
    -4*x(3) - 5*x(2) - 2*x(1) % x3' = y'''
];
```

What MATLAB is seeing

- x(1) → current value of y
- x(2) → current value of y'
- x(3) → current value of y''

The function returns:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix}$$

Which is the requirement of ode45.

Time span and initial conditions

```
tspan = [0 10];
```

Solve from $t = 0$ to $t = 10$.

```
x0 = [1; 0; 0];
```

This means:

- $y(0) = 1$
- $y'(0) = 0$
- $y''(0) = 0$

Number of initial conditions = order of ODE

Solve

```
[t, x] = ode45(odefun, tspan, x0);
```

- t → time points chosen adaptively
- x(:,1) → y(t)
- x(:,2) → y'(t)
- x(:,3) → y''(t)

Plot the physical variable

```
plot(t, x(:,1), 'LineWidth', 2)
grid on
xlabel('Time')
ylabel('y(t)')
```

Common Mistakes

Below is a list of common mistakes encountered while solving higher order ODEs using ode45:

Mistakes	Correct Approach
Mixing up the state order, assigning x(1) and x(2) incorrectly.	Always keep: • x(1) = variable • x(2) = first derivative • x(3) = second derivative …..so on
Insufficient initial conditions.	Number of ICs must match the order of ODE.
Wrong Output Shape, returning scalar instead of vector.	ode45 expects column vector.

These errors often lead to completely wrong results, sometimes without any obvious MATLAB error message, which makes debugging very difficult.

Comparison of Methods

We have discussed about Euler's method, RK2, RK4, and ode45 to solve ODEs. A comparison among these methods is presented as follows:

Method	Order	Step Type	Adaptive
Euler	1	Fixed	No
RK2	2	Fixed	No
RK4	4	Fixed	No
ode45	4/5	Adaptive	Yes

We are going to solve an ODE with RK2, RK4 and ode45 to identify the best method. We will also compare the results with the exact solution.

Consider:

$$\frac{dy}{dt} = -2y, y(0) = 1$$

$$h = 0.2$$

Exact solution:

$$y(t) = e^{-2t}$$

We will solve on:

$$t \in [0,2]$$

MATLAB Implementation

```
% ODE and exact solution
f = @(t,y) -2*y;
y_exact = @(t) exp(-2*t);

% Time settings
h = 0.2;
t = 0:h:2;
y0 = 1;
```

```matlab
% Preallocate
y_rk2 = zeros(size(t)); y_rk2(1) = y0;
y_rk4 = zeros(size(t)); y_rk4(1) = y0;

% -----------------------
% RK2 and RK4 together
% -----------------------
for i = 1:length(t)-1

    % ----- RK2 -----
    k1 = f(t(i), y_rk2(i));
    k2 = f(t(i)+h, y_rk2(i)+h*k1);
    y_rk2(i+1) = y_rk2(i) + (h/2)*(k1 + k2);

    % ----- RK4 -----
    k1 = f(t(i), y_rk4(i));
    k2 = f(t(i)+h/2, y_rk4(i)+(h/2)*k1);
    k3 = f(t(i)+h/2, y_rk4(i)+(h/2)*k2);
    k4 = f(t(i)+h, y_rk4(i)+h*k3);
    y_rk4(i+1) = y_rk4(i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);

end

% ode45
[t45, y45] = ode45(f, [0 2], y0);

% Exact values
y_ex = y_exact(t);

% -----------------------
% Plot
% -----------------------
plot(t, y_ex, 'k-', ...
     t, y_rk2, 'ro--', ...
     t, y_rk4, 'bs--', ...
     t45, y45, 'g', 'LineWidth', 1.5)

grid on
xlabel('Time')
ylabel('y(t)')
title('Exact vs RK2 vs RK4 vs ode45')
legend('Exact','RK2','RK4','ode45')

% -----------------------
% Errors
% -----------------------
err_rk2 = max(abs(y_rk2 - y_ex));
err_rk4 = max(abs(y_rk4 - y_ex));
err_45  = max(abs(y45 - y_exact(t45)));

fprintf('Max Error RK2   = %.5e\n', err_rk2)
fprintf('Max Error RK4   = %.5e\n', err_rk4)
fprintf('Max Error ode45 = %.5e\n', err_45)
```

Exercises

1. Solve the IVP using RK2 (Heun), RK4 Method on on $t \in [0,1]$, h = 0.1. Plot both solutions on the same graph.

$$\frac{dy}{dt} = 3y, y(0) = 1$$

2. Solve using ode45

$$y'' + y = \cos(t)$$

with:

$$y(0) = 0, y'(0) = 1; \text{time span} = [0,10]$$

3. Solve the fourth-order differential equation using ode45:

$$y^{(4)} - 2y'' + y = 0$$

with initial conditions:

$$y(0) = 1, y'(0) = 0, y''(0) = 0, y'''(0) = 1$$

on the interval:

$$t \in [0,5]$$

Experiment 8: Methods for Solving Nonlinear Equations in MATLAB

Objectives

After completing this experiment, students will be able to:

- Understand the theory behind common root-finding methods
- Formulate the mathematical basis of each method
- Implement nonlinear equation solvers in MATLAB
- Compare convergence behavior of different methods

1. Introduction

Nonlinear equations arise frequently in engineering, science, and applied mathematics. A nonlinear equation is generally written as:

$$f(x) = 0$$

Where $f(x)$ is a nonlinear function of the variable x. Analytical solutions are often not possible, so numerical (iterative) methods are used to approximate the roots. This experiment introduces commonly used numerical methods for solving nonlinear equations and demonstrates their implementation in MATLAB.

Such nonlinear equations frequently arise in areas such as heat transfer, fluid mechanics, structural analysis, control systems, manufacturing processes, supply chain modeling, and optimization problems..

A value $x = \alpha$ is called a **root** (or solution) of the equation if:

$$f(\alpha) = 0$$

In most practical situations, the exact value of α cannot be obtained in closed form. Therefore, **numerical (iterative) methods** are employed to compute an approximate solution $\tilde{x}$ such that:

$$|f(\tilde{x})| \leq \varepsilon$$

Where ε is a prescribed tolerance representing an acceptable level of numerical error.

1.1 Iterative Nature of Nonlinear Equation Solvers

Numerical methods for solving nonlinear equations are inherently **iterative**. Starting from an initial guess (or an initial interval), a sequence of approximations is generated:

$$x_o, x_1, x_2, \dots \dots .. x_k$$

The objective of the method is to ensure that the sequence converges to the true root:

$$\lim_{k \to \infty} x_k = \alpha$$

Each numerical technique defines a specific **iteration formula** that computes x_{k+1} using previously computed values.

1.2 Classification of Root-Finding Methods

Numerical methods for solving nonlinear equations can be broadly classified into two main categories:

(a) Bracketing Methods

Bracketing methods require an initial interval $[a, b]$such that:

$$f(a)f(b) < 0$$

This condition guarantees the existence of at least one root in the interval $[a, b]$ due to the **Intermediate Value Theorem** which states, if a function $f(x)$ is **continuous** on a closed interval $[a, b]$, and if $f(a)$ and $f(b)$have **opposite signs**, then there exists **at least one value** c in the interval (a, b) such that:

$$f(c) = 0$$

These methods are generally **robust and reliable**, but they converge relatively slowly.

Examples:

- Bisection Method
- False Position (Regula Falsi) Method

(b) Open Methods

Open methods do not require the root to be bracketed. Instead, they start from one or more initial guesses and use function behavior to approach the root. These methods typically converge faster but may diverge if the initial guess is not chosen properly.

Examples:

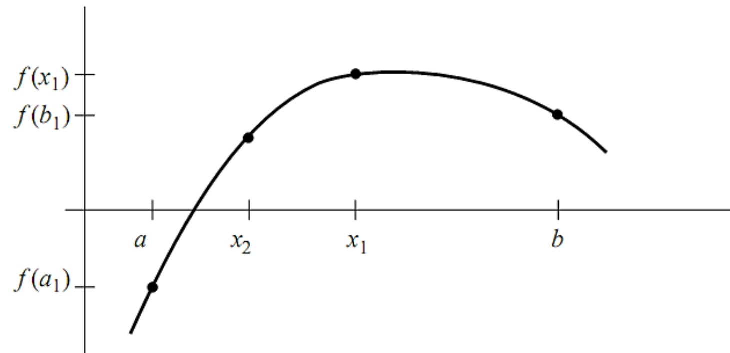
- Fixed Point Iteration Method
- Newton–Raphson Method
- Secant Method

2. Bisection Method

The Bisection Method is a bracketing method based on the Intermediate Value Theorem. If a continuous function $f(x)$ satisfies

$$f(a)f(b) < 0$$

then at least one root exists in the interval $[a, b]$.



Mathematical Formulation

I. Select a range of x values (say x_l and x_u), such that $f(x_l) \times f(x_u)$<0
II. New approximation of the root is computed as:

$$x_r = \frac{x_l + x_u}{2}$$

III. Apply some test:
- if $f(x_l) \times f(x_r)$<0 then set $x_u = x_r$ and check the stopping criteria
- if $f(x_l) \times f(x_r)$>0, then set $x_l = x_r$ and check the stopping criteria
- if $f(x_l) \times f(x_r)$=0, then set root =x_r and stop

IV. Stopping criteria:

If approximate percent relative error $\varepsilon_a \leq \varepsilon_s$ $(stopping\ criteria)$ then stop, where

$$\varepsilon_a = \left|\frac{x_{rnew} - x_{rold}}{x_{rnew}}\right| \times 100\%$$

Example

Solve:

$$f(x) = x^3 - x - 2$$

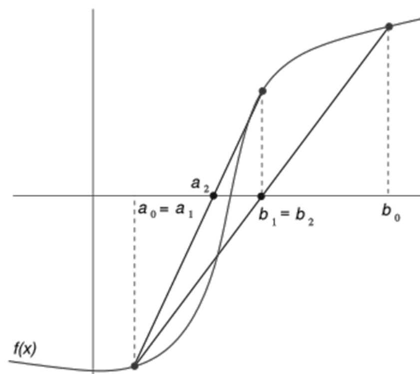
Choose $x_l = 1$ and $x_u = 2$

```
function s = func(x)
s = x^3 - x - 2;
end
```

```
xl = input('Enter the lower value, xl = ');
xu = input('Enter the upper value, xu = ');
elimit = 0.0001;
xold = 0.0;

if func(xl)*func(xu) > 0
    disp('There is no root in this range');
else
    for i = 1:100
        xr = (xl + xu)/2;

        if func(xr)*func(xl) > 0
            xl = xr;
        elseif func(xr)*func(xl) < 0
            xu = xr;
        else
            break
        end

        if abs((xr - xold)/xr) <= elimit
            break
        end

        xold = xr;
    end

    disp('Root is:');
    disp(xr);
    disp('Number of iterations:');
    disp(i);
end
```

3. False Position (Regula Falsi) Method

The False Position Method is also a bracketing method, but instead of the midpoint, it uses linear interpolation between (a, f(a)) and (b, f(b)).



Mathematical Formulation

I. Select a range of x values (say x_1 and x_2), such that $f(x_1) \times f(x_2)<0$
II. New approximation of the root is computed as:

$$x_3 = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)}$$

III. Apply some test:
- if $f(x_1) \times f(x_3)<0$ then set $x_2 = x_3$ and check the stopping criteria
- if $f(x_1) \times f(x_3)>0$, then set $x_1 = x_3$ and check the stopping criteria
- if $f(x_1) \times f(x_3)=0$, then set root $=x_3$ and stop

IV. Stopping criteria:

If $\varepsilon_a \leq \varepsilon_s$ then stop, where

$$\varepsilon_a = \left| \frac{x_{3new} - x_{3old}}{x_{3new}} \right| \times 100\%$$

Example

Solve:

$$f(x) = x^3 - x - 2$$

```
function s = func(x)
s = x^3 - x - 2;
end
```

```
x1 = input('Enter the lower value, x1 = ');
x2 = input('Enter the upper value, x2 = ');
elimit = 0.0001;
xold = 0.0;

if func(x1)*func(x2) > 0
    disp('There is no root in this range');
else

    for i = 1:100
        x3 = (x1*func(x2) - x2*func(x1)) / (func(x2) - func(x1));

        if func(x3)*func(x1) < 0
           x2 = x3;
        elseif func(x3)*func(x1) > 0
           x1 = x3;
        else
           break
        end

       if abs((x3 - xold)/x3) <= elimit
          break
```

```
        end

      xold = x3;
    end
disp('Root is:');
disp(x3);
disp('Number of iterations:');
disp(i);
end
```

4. Fixed Point Iteration Method

In Fixed Point Iteration, the equation $f(x) = 0$ is rewritten as:

$$x = g(x)$$

This transformation can be accomplished either by algebraic manipulation or by simply adding x to both sides of the original equation. The method converges if ($|g'(x)| < 1$) near the fixed point.

Mathematical Formulation

The iterative scheme is:

$$x_{k+1} = g(x_k)$$

Example

Given:

$$f(x) = x^3 - x - 2$$

Rewrite as:

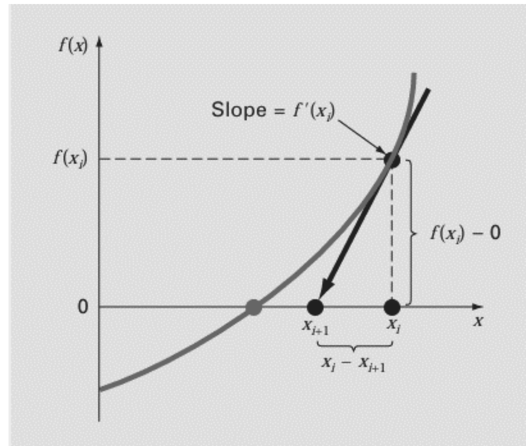
$$x = g(x) = (x + 2)^{1/3}$$

```
x0 = input('Enter the initial guess, x0 = ');
elimit = 0.0001;

for i = 1:100
    x1 = (x0 + 2)^(1/3);

    if abs((x1 - x0)/x1) <= elimit
        break
    end

    x0 = x1;
end
```

```
disp('Root is:');
disp(x1);
disp('Number of iterations:');
disp(i);
```

5. Newton–Raphson Method

The Newton–Raphson method uses a first-order Taylor series expansion and requires the derivative of $f(x)$.



Mathematical Formulation

I. Start at the point $(x_1, f(x_1))$

II. If $f'(x_1)$ is $\neq 0$ the new root will be found:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

III. Examine if $f(x_2) = 0$ or $abs\left\{\frac{(x_2 - x_1)}{x_2}\right\} \leq Error\ limit$

IV. If yes, solution $x_r = x_2$. If not, $x_1 = x_2$, repeat from step II.

Example

Solve: $f(x) = x^3 - x - 2$

```
function y=func(x)
y= x^3-x-2;
end
```

```
function s = df(x)
s = 3*x^2-1;
end
```

```
x0 = input('Enter the initial guess, x0 = ');
elimit = 0.0001;

for i = 1:100

   if df(x0) == 0
      disp('Method fails: zero derivative');
      break
    end

    x1 = x0 - f(x0)/df(x0);

    if abs((x1 - x0)/x1) <= elimit
       break
    end

    x0 = x1;
end
disp('Root is:');
disp(x1);
disp('Number of iterations:');
disp(i);
```

6. Secant Method

The Secant Method is similar to Newton–Raphson but avoids explicit derivative calculation by using finite differences.

Mathematical Formulation:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

Example: Solve: $f(x) = x^3 - x - 2$

```
function s = func(x)
s = x^3-x-2
end
```

```
x0 = input('Enter first initial guess, x0 = ');
x1 = input('Enter second initial guess, x1 = ');
elimit = 0.0001;


for i = 1:100
    if func(x0) == func(x1)
       disp('Secant method fails: division by zero)');
       break
    end

    x2 = x1 - ((func(x1)*(x0 - x1))/(func(x0) - func(x1)));
```

```
    if abs((x2 - x1)/x2) <= elimit
        break
    end

    x0 = x1;
    x1 = x2;
end
    disp('Root is:');
    disp(x2);
    disp('Number of iterations:');
    disp(i);
```

7. Interpolation Method (Polynomial Root Approximation)

Interpolation methods approximate the function using a polynomial based on discrete data points and then solve the polynomial equation.

Mathematical Formulation

Given data points $x_i, y_i)$ construct an interpolating polynomial $P(x)$ such that:

$$P(x_i) = y_i$$

Roots of $P(x)$ approximate the roots of $f(x)$.

Example

Given sampled values of:

$$f(x) = x^3 - x - 2$$

```
x = [1 1.5 2];
y = x.^3 - x - 2;

p = polyfit(x, y, 2);  % quadratic interpolation
roots_p = roots(p)
```

8. Müller’s Method

Müller’s Method is an **open, iterative numerical method** for solving nonlinear equations. It is based on **quadratic interpolation** using three successive approximations. Unlike the Newton–Raphson method, it does not require derivatives, and unlike bracketing methods, it does not require an interval containing the root.

An important advantage of Müller’s Method is that it can converge to **complex roots**.

Mathematical Formulation:

Given three approximations x_{k-2}, x_{k-1}, x_k, a quadratic polynomial is fitted through:

$$(x_{k-2}, f(x_{k-2})), \qquad (x_{k-1}, f(x_{k-1})), \qquad (x_k, f(x_k))$$

The quadratic polynomial is written as:

$$P(x) = a(x - x_k)^2 + b(x - x_k) + c$$

where

$$c = f(x_k)$$

$$b = \frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k}$$

$$a = \frac{\left(\frac{f(x_{k-2}) - f(x_k)}{x_{k-2} - x_k}\right) - b}{x_{k-2} - x_{k-1}}$$

The next approximation is:

$$x_{k+1} = x_k + \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

It yields two roots, corresponding to the ± term in the denominator. In Müller's method, the sign is chosen to agree with the sign of b. This choice will result in the largest denominator, and hence, will give the root estimate that is closest to x_k.

Error can be calculated as: $abs\left\{\left|\frac{(x_{k+1} - x_k)}{x_{k+1}}\right|\right\}$

Example

Solve:

$$f(x) = x^3 - x - 2$$

```
function s = func(x)
s = x^3-x-2
end
```

```
% Muller's Method

xkm2 = input('Enter x(k-2)  = ');
xkm1 = input('Enter x(k-1)  = ');
xk   = input('Enter x(k)    = ');

elimit = 0.0001;
xold = 0.0;

for i = 1:100
    c = func(xk);
    b = (func(xkm1) - func(xk)) / (xkm1 - xk);
    a = ( (func(xkm2) - func(xk)) / (xkm2 - xk) - b ) / (xkm2 - xkm1);

    D = sqrt(b^2 - 4*a*c);

    if abs(b + D) > abs(b - D)
        xkp1 = xk + (-2*c)/(b + D);
    else
        xkp1 = xk + (-2*c)/(b - D);
    end

    if abs((xkp1 - xold)/xkp1) <= elimit
        break
    end

    xkm2  = xkm1;
    xkm1  = xk;
    xk    = xkp1;
    xold = xkp1;
end

disp('Root is:')
disp(xkp1)
disp('Number of iterations:')
disp(i)
```

9. MATLAB in-built Functions

fzero: fzero is a built-in MATLAB function used to **find a real root of a single-variable nonlinear equation.** It is a **numerical root-finding function** that internally uses a combination of **bisection, secant, and inverse quadratic interpolation methods** to ensure robustness and fast convergence.

Syntax of fzero:

```
x = fzero(fun, x0)
or
x = fzero(fun, [a b])
```

Where:

- `fun` → function handle or function name
- `x0` → initial guess

- `[a b]` → interval that brackets a root
- `x` → computed root

roots: The roots function is used to find **all roots (real and complex)** of a **polynomial equation**.
Syntax of roots:
x = roots(p)
Where:

- p → row vector of polynomial coefficients
- x → vector containing all roots

Solve:

- $x^3 - 5x^2 - x + 2 = 0$

```
p = [1 -5 -1 2];
r = roots(p)
```

Comparison of Methods

Method	**Bracketing**	**Derivative Required**	**Convergence Speed**
Bisection	Yes	No	Slow (Linear)
False Position	Yes	No	Linear
Fixed Point	No	No	Linear
Newton–Raphson	No	Yes	Quadratic
Secant	No	No	Superlinear

Exercise:

Problem-1:

In environmental engineering (a specialty area in renewable energy) the following equation can be used to compute the oxygen level c (mg/L) in a river downstream from a sewage discharge:

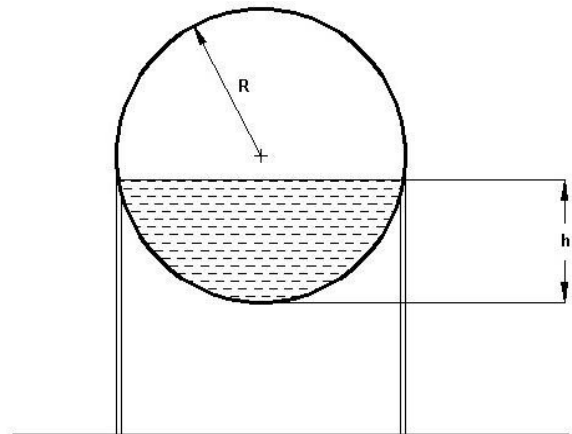
$$c = 10 - 20(e^{-0.15x} - e^{-0.5x})$$

where x is the distance downstream in kilometers. Determine the distance downstream where the oxygen level first falls to a reading of 5 mg/L. Determine your answer to a 1% error. Use **Regula Falsi / False Position Method**.

Problem-2: A resistor, capacitor and inductor are connected in parallel. Z = impedance (Ω) and ω = angular and ω = angular frequency. Find the ω that results in the impedance of 100 Ω using **false position method** with initial guesses of 1 and 1000 for the following parameters, R = 225 Ω, L = 0.5 H, C = 0.6×10^{-6} H. Use the following equation: $\frac{1}{Z} = \sqrt{\frac{1}{R^2} + (\omega C - \frac{1}{\omega L})^2}$

Problem-3:

You are designing a spherical tank to hold water for a small village in a developing country. The volume of liquid it can hold can be computed as-



$V=\pi h^2\times(3R-h) \times (2/3)$

Where V=volume [m^3],

h=depth of water in tank [m],

R=the tank radius [m].

If R=3m, to what depth must the tank be filled so that it holds $30m^3$ water.

Use **Newton-Raphson** Technique. **[ε_s = 0.05%].**

Experiment 9: Basic plotting and Curve Fitting

Objective

The main objectives of this Experiment are:

- To understand graphical visualization and data representation using MATLAB
- To plot and analyze mathematical functions and experimental data using 2D and 3D graphs
- To apply curve fitting techniques in MATLAB for analyzing engineering and scientific data

Introduction

Graph plotting and curve fitting are essential techniques in engineering and science for analyzing and interpreting data. MATLAB provides extensive tools for plotting, allowing visualization of trends, comparison of experimental and theoretical results, and validation of models.

Curve fitting helps find a mathematical relationship that best approximates a set of data points. Real-world data often contains noise, so an approximate fit is preferred over an exact one. MATLAB functions and the Curve Fitting Toolbox make linear, nonlinear, and polynomial fitting efficient and accurate.

Graph Plotting

MATLAB uses vectors and matrices as basic data structures. To plot a graph, data must be stored in vector form. The plot function connects corresponding elements of two vectors (x and y) to create a graphical representation.

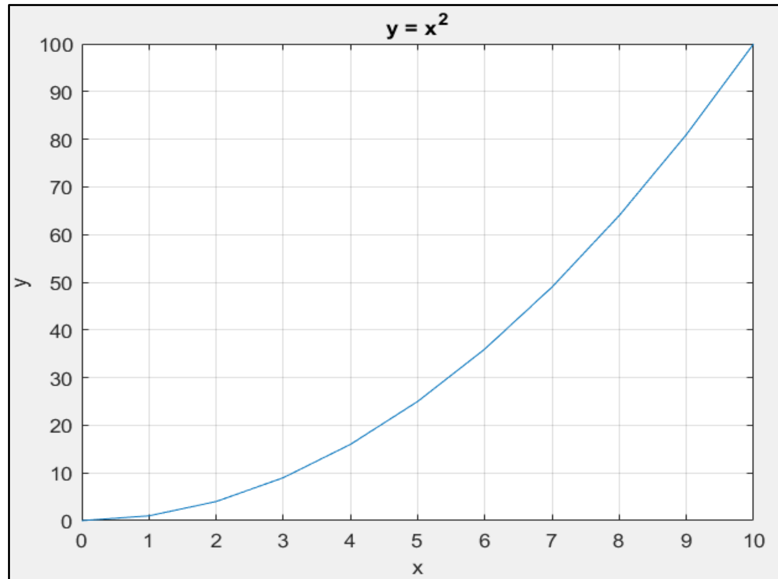
General Syntax

```
plot(x, y)
```

Plotting 2D line graphs

Example: Plot the equation $y = x^2$ for x ranging from 0 to 10

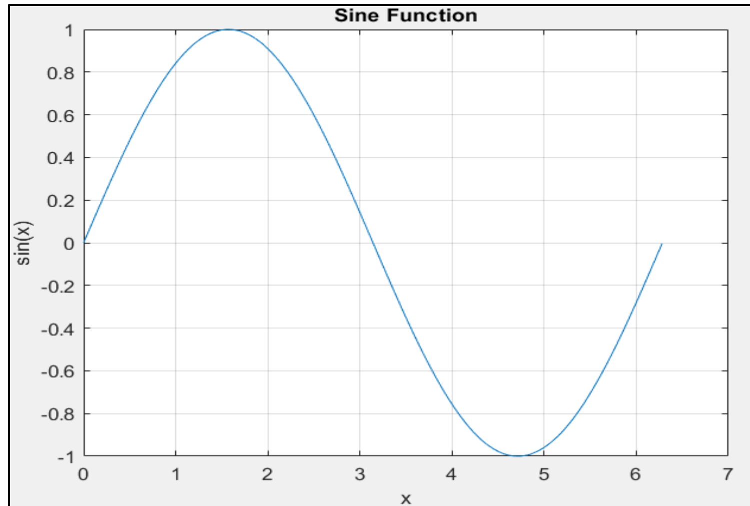
```
x = 0:1:10;          % Independent variable
y = x.^2;            % Dependent variable
plot(x, y);
xlabel('x');
ylabel('y');
title('y = x^2');
grid on;
```



Plotting Mathematical Functions

Example: Plot a sine wave between 0 and 2π.

```
x = 0:0.01:2*pi;
y = sin(x);
plot(x, y);
xlabel('x');
ylabel('sin(x)');
title('Sine Function');
grid on;
```



Graph customization

Various line types, plot symbols and colors may be obtained with **plot(X, Y, S)** where **S** is a character string made from one element from any or all the following 3 columns:

Color	Meaning	Symbol	Meaning	Line Type	Meaning
b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		

		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

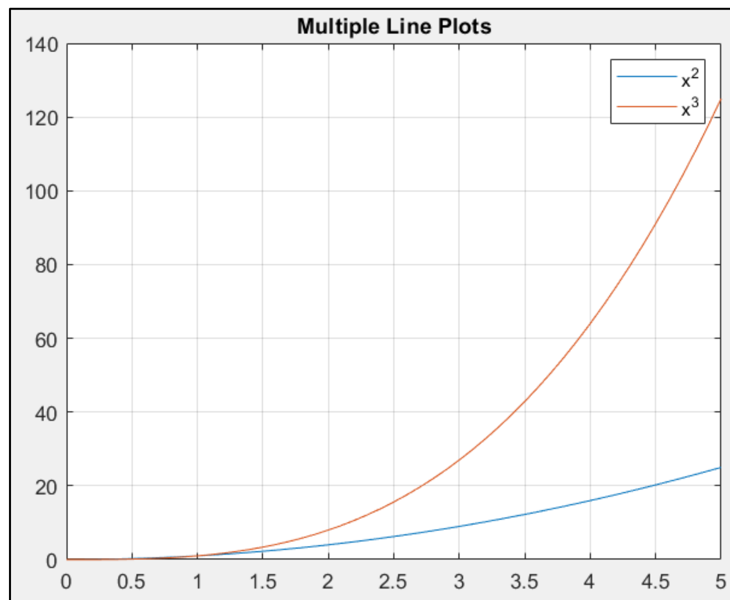
Example: Plot a cosine wave between 0 and 2π.

```
x = 0:0.1:2*pi;
y = cos(x);
plot(x, y, 'r-->', 'Linewidth', 2);
grid on;
xlabel('x');
ylabel('cos(x)');
title('Customized Plot');
```

Multiple plots

Example: Plotting two mathematical functions ($y = x^2$ and $y = x^3$) on the same graph.

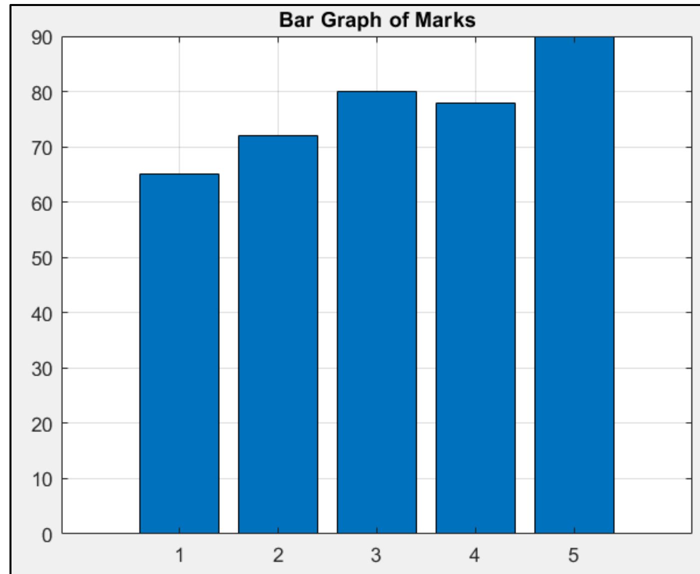
```
x = 0:0.1:5;
y1 = x.^2;
y2 = x.^3;
plot(x, y1);
hold on;
plot(x, y2);
hold off;
grid on;
legend('x^2', 'x^3');
title('Multiple Line Plots');
```



Bar Chart

Example: Generate a bar chart for the data marks = [65 72 80 78 90].

```
marks = [65 72 80 78 90];
bar(marks);
grid on;
title('Bar Graph of Marks');
```



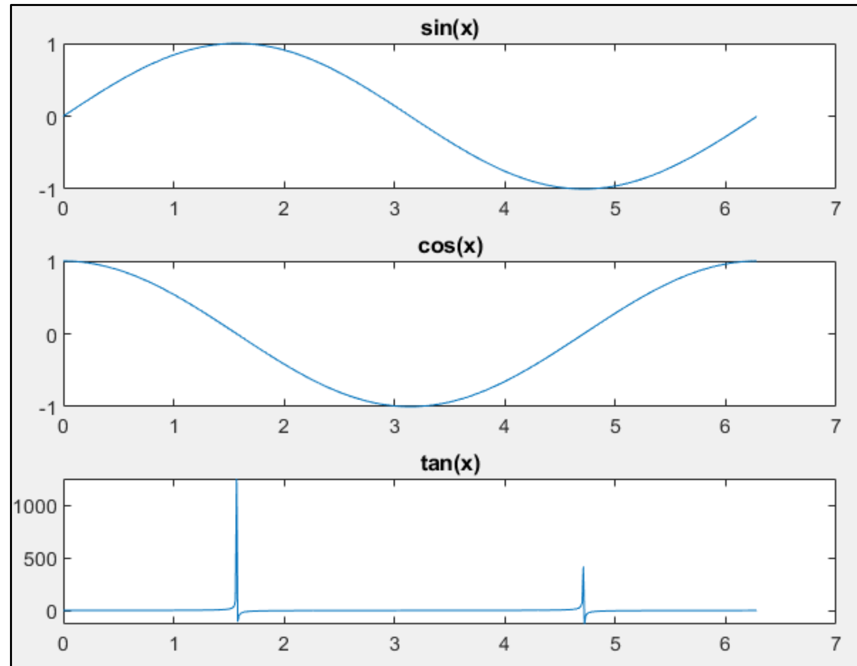
subplot

Subplot is used to create multiple plotting areas within a single figure window. The command subplot(m,n,p) (or subplot(mnp)) divides the figure into an m × n grid of smaller axes and selects the p-th position for the current plot.

Example:

Generate three subplots for the functions sin(x), cos(x), and tan(x) over the range 0 to 2π.

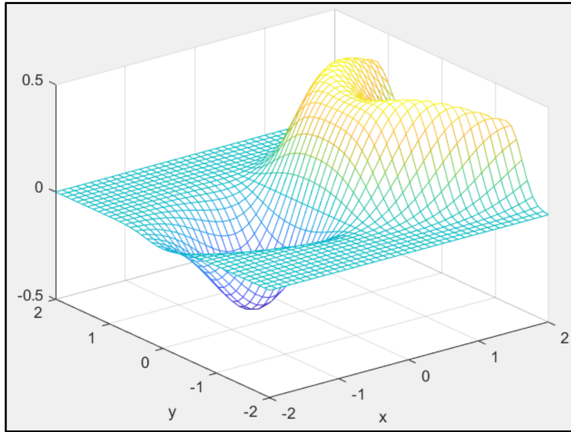
```
x = 0:0.01:2*pi;
subplot(3,1,1);
plot(x, sin(x));
title('sin(x)');

subplot(3,1,2);
plot(x, cos(x),);
title('cos(x)');

subplot(3,1,3);
plot(x, tan(x));
title('tan(x)');
```



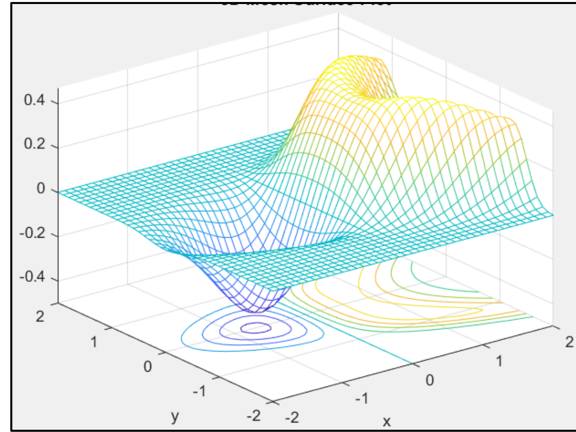
<u>Plotting 3D line graphs</u>

3D Plotting Functions in MATLAB

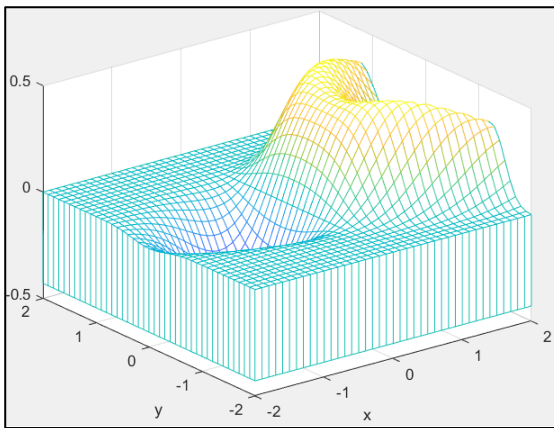
Function	Description
plot3(x,y,z)	Creates a 3D line plot
mesh(x,y,z)	Creates a wireframe 3D surface
meshc(x,y,z)	Mesh with contour plot underneath
meshz(x,y,z)	Mesh with vertical reference lines
surf(x,y,z)	Shaded 3D surface plot
surfc(x,y,z)	Surface with contour plot underneath
contour(x,y,z)	2D contour plot
waterfall(x,y,z)	3D plot with mesh lines in one direction
meshgrid(x,y)	Creates grid matrices for surface plotting



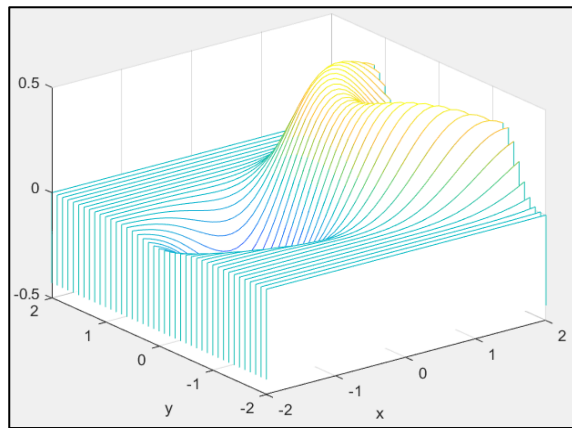
mesh



meshc



meshz



waterfall

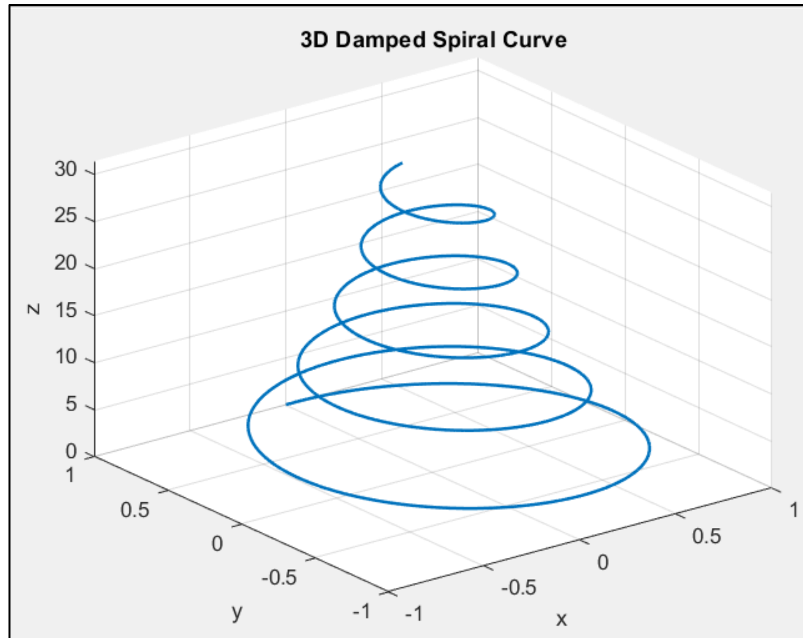
Example:

Plot a three-dimensional damped spiral curve defined by the parametric equations

$$x = e^{-0.05t}\sin(t), y = e^{-0.05t}\cos(t), z = t$$

where the parameter t varies from 0 to 10π with a step size of $\pi/50$.

```
clc, clear;
t = 0:pi/50:10*pi;
x = exp(-0.05*t).*sin(t);
y = exp(-0.05*t).*cos(t);
z = t;
plot3(x, y, z, 'Linewidth', 1.5);
grid on;
xlabel('x');
ylabel('y');
zlabel('z');
title('3D Damped Spiral Curve');
```



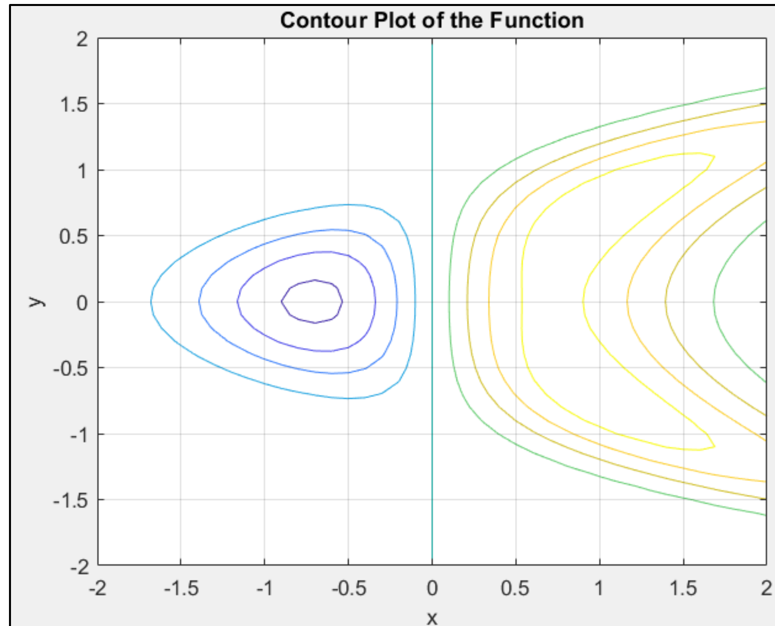
Example:

Write a MATLAB program to draw a contour plot of the function

$$z = x\, e^{-[(x-y^2)^2+y^2]}$$

over the range $-2 \le x \le 2$ and $-2 \le y \le 2$.

```
x = -2:0.1:2;
y = -2:0.1:2;

[X, Y] = meshgrid(x, y);
Z = X.* exp(-((X - Y.^2).^2 + Y.^2));

contour(X, Y, Z)
xlabel('x')
ylabel('y')
title('Contour Plot of the Function')
grid on;
```



Interpolation

Interpolation is used to estimate data points between two known points. MATLAB has interpolation functions. In one-dimensional interpolation, each point has one independent variable (x) and one dependent variable (y). In two-dimensional interpolation, each point has two independent variables (x and y) and one dependent variable (z).

One-dimensional interpolation:

One-dimensional interpolation in MATLAB is done with the interp1 (the last character is the number one) function, which has the form:

$$\boldsymbol{yi = interp1(x, y, xi, 'method')}$$

yi is the interpolated value

Method of interpolation, typed as a string (optional)

x is a vector with the horizontal coordinates of the input data points (independent variable)
y is a vector with the vertical coordinates of the input data points (dependent variable)
xi is the horizontal coordinate of the interpolation point (independent variable)
Method of interpolation, typed as a string (optional)

MATLAB can do the interpolation using one of several methods that can be specified. These methods include:

- **‘nearest’** returns the value of the data point that is nearest to the interpolated point.
- **‘linear’** uses linear spline interpolation.
- **‘spline’** uses cubic spline interpolation.
- **‘pchip’** uses piecewise cubic Hermite interpolation, also called ‘cubic.’

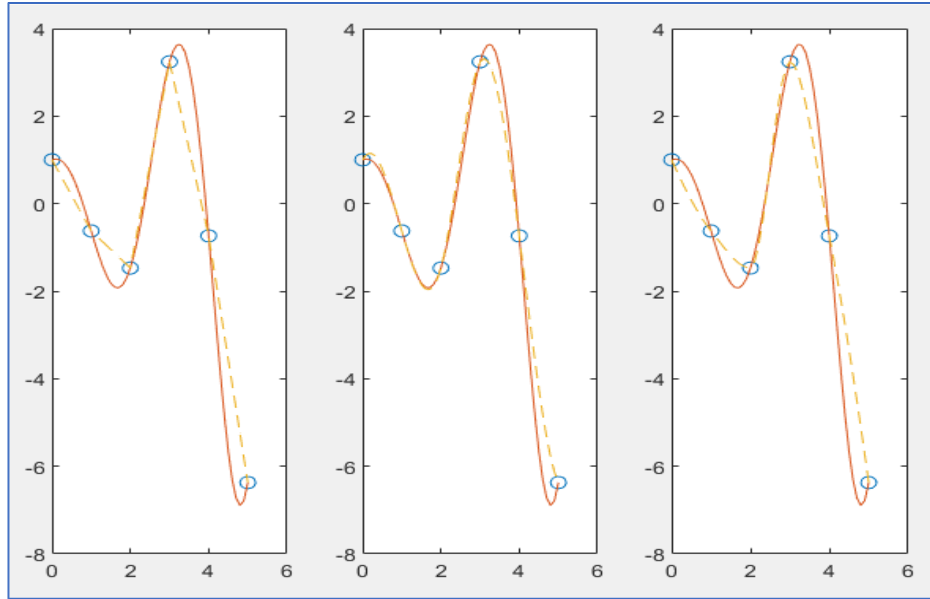
Example

The following data points, which are points of the function f(x)=1.5xcos(2x), are given. Use linear, spline, and pchip interpolation methods to calculate the value of y between the points. Make a figure for each of the interpolation methods. In the figure show the points, a plot of the function, and a curve that corresponds

Given Data

x	0	1	2	3	4	5
y	1.0	−0.6242	−1.4707	3.2406	−0.7366	−6.3717

```
x = 0:1:5;
y = [1.0 -0.6242 -1.4707 3.2406 -0.7366 -6.3717];

% Create vector xi with points for interpolation
xi = 0:0.1:5;

% Calculate y points from linear interpolation
yilin = interp1(x, y, xi, 'linear');

% Calculate y points from spline interpolation
yispl = interp1(x, y, xi, 'spline');

% Calculate y points from pchip interpolation
yipch = interp1(x, y, xi, 'pchip');

% Calculate y points from the function
yfun = 1.5.^xi .* cos(2*xi);

% Linear interpolation plot
subplot(1,3,1)
plot(x, y, 'o', xi, yfun, xi, yilin, '--');
% Spline interpolation plot
subplot(1,3,2)
plot(x, y, 'o', xi, yfun, xi, yispl, '--');

% PCHIP interpolation plot
subplot(1,3,3)
plot(x, y, 'o', xi, yfun, xi, yipch, '--');
```



Interpolation functions in MATLAB:

Function	**Description**
interp1(x, y, xi, method)	One-dimensional interpolation; estimates values at intermediate points along a line.
interp2(X, Y, Z, XI, YI, method)	Two-dimensional interpolation; estimates values on a grid, like surfaces.
interp3(x, y, z, V, xi, yi, zi, method)	Three-dimensional interpolation; estimates values inside a 3D volume.

Curve fitting

Curve fitting is a numerical technique used to determine a mathematical model that best represents a given set of experimental data points. When measured data contains errors or noise, curve fitting approximates the overall trend of the data rather than forcing the curve to pass through every point, which distinguishes it from interpolation. MATLAB provides built-in functions to create empirical data models based on the least-squares method, ensuring that the overall error between the experimental data and the fitted curve is minimized.

Curve Fitting with Polynomials

MATLAB has two functions, polyfit and polyval, which can quickly and easily fit a polynomial to a set of data points. A first order polynomial is the linear equation that best fits the data. A polynomial can also be used to fit the data in a quadratic sense. As a reminder, the general formula for a polynomial is:

$$f(x) = a_0 x^N + a_1 x^{N-1} + a_2 x^{N-2} + \cdots + a_{N-1} x + a_N$$

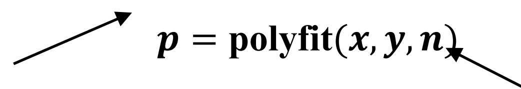
The degree of a polynomial is equal to the largest value of the exponents of the independent variable, x.

For example :

- first degree equation: $y = 2x + 1$
- second degree equation: $y = 3x^2 + 2x$
- third degree equation: $y = 3x^3 + 8x^2 + 4x + 5$

Polyfit and Polyval

Curve fitting with polynomials is done in MATLAB with the **polyfit** function, which uses the least squares method. Polyfit actually generates the coefficients of the polynomial (which can be used to simulate a curve to fit the data) according to the degree specified. The basic form of the polyfit function is:

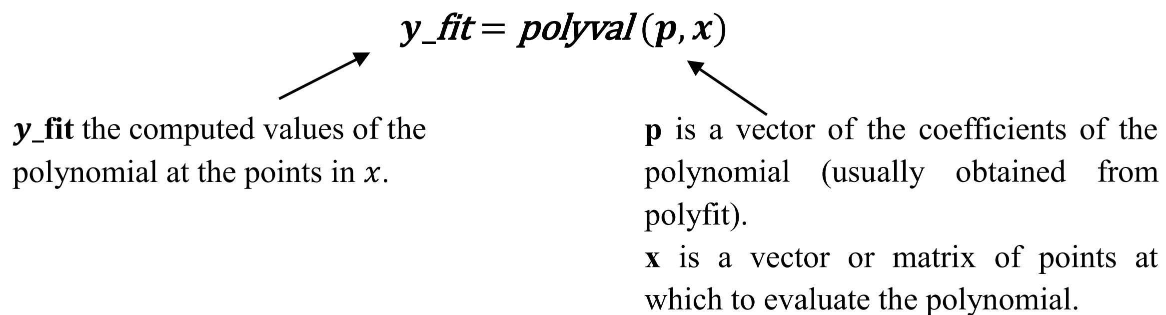


p is the vector of the coefficients of the polynomial that fits the data.

x is a vector with the horizontal coordinates of the data points (independent variable).
y is a vector with the vertical coordinates of the data points (dependent variable).
n is the degree of the polynomial.

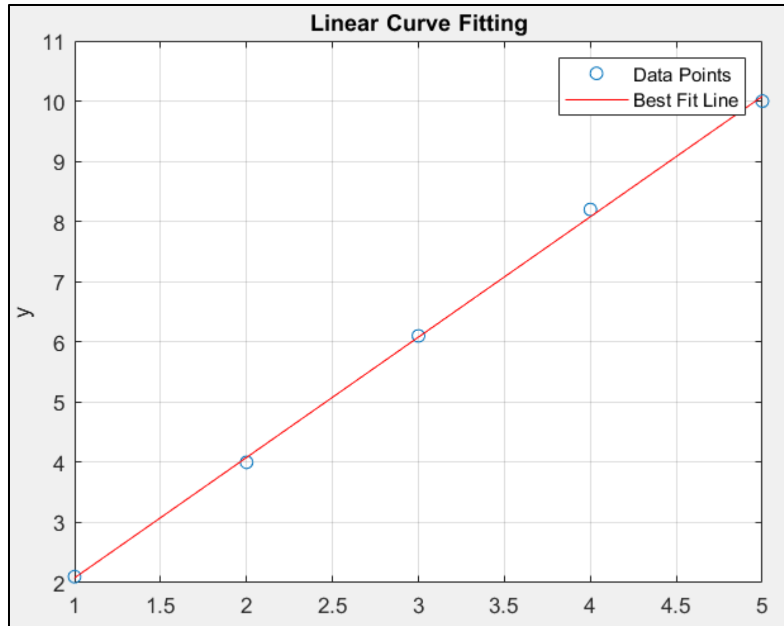
For the same set of m points, the polyfit function can be used to fit polynomials of any order up to m-1. If n = 1 the polynomial is a straight line, if n = 2 the polynomial is a parabola, and so on. The polynomial passes through all the points if n= m -1 (the order of the polynomial is one less than the number of points). It should be pointed out here that a polynomial that passes through all the points, or polynomials with higher order, do not necessarily give a better fit over all. High-order polynomials can deviate significantly between the data points.

"Polyval" evaluates a polynomial for a given set of x values. So, polyval actually generates a curve to fit the data based on the coefficients found using polyfit.The basic form of the polyval function is:



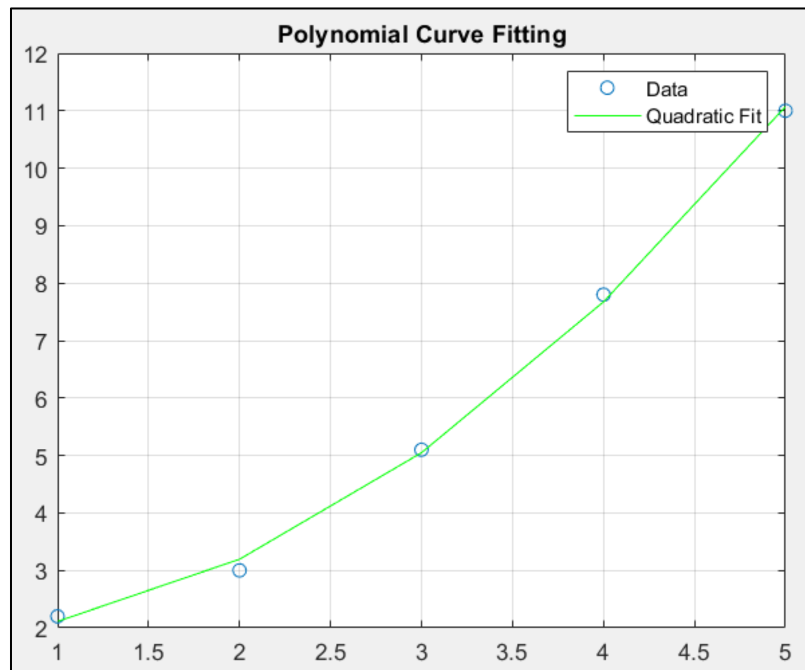
Example

```
x = [1 2 3 4 5];
y = [2.1 4.0 6.1 8.2 10.0];
p = polyfit(x, y, 1);   % Linear fit
y_fit = polyval(p, x);
plot(x, y, 'o', x, y_fit, '-r');
grid on;
xlabel('x');
ylabel('y');
title('Linear Curve Fitting');
legend('Data Points','Best Fit Line');
```



Example:

```
x = [1 2 3 4 5];
y = [2.2 3.0 5.1 7.8 11.0];
p = polyfit(x, y, 2);         % Quadratic fit
y_fit = polyval(p, x);
plot(x, y, 'o', x, y_fit, '-g');
grid on;
title('Polynomial Curve Fitting');
legend('Data','Quadratic Fit');
```



Curve Fitting with Functions Other than Polynomials

Any situation in science and engineering requires fitting functions that are not polynomials to given data. Theoretically, any function can be used to model data within some range. For a particular data set, however, some functions provide a better fit than others. In addition, determining the best-fitting coefficients can be more difficult for some functions than for others. This section covers curve fitting with power, exponential, logarithmic, and reciprocal functions, which are commonly used. The forms of these functions are:

$y = bx^m$(power function)

$y = be^{mx}$ or $y = b10^{mx}$(exponential function)

$y = m\ln(x) + b$ or $y = m\log(x) + b$(logarithmic function)

$y = \frac{1}{mx + b}$(reciprocal function)

All of these functions can easily be fitted to given data with the polyfit function. This is done by rewriting the functions in a form that can be fitted with a linear polynomial ($n = 1$), which is

$$y = mx + b$$

The logarithmic function is already in this form, and the power, exponential and reciprocal equations can be rewritten as:

$\ln(y) = m\ln(x) + \ln(b)$ (power function)

$\ln(y) = mx + \ln(b)$ or $\log(y) = mx + \log(b)$(exponential function)

$\frac{1}{y} = mx + b$(reciprocal function)

These equations describe a linear relationship between $\ln(y)$ and $\ln(x)$ for the power function, between $\ln(y)$ and x for the exponential function, between y and $\ln(x)$or $\log(x)$for the logarithmic function, and between $\frac{1}{y}$and x for the reciprocal function. This means that the polyfit(x, y, 1) function can be used to determine the best-fit constants m and b for best fit if, instead of x and y, The transformed variables are used.

Using MATLAB polyfit for Nonlinear Fitting

Function Type	Model Formula	MATLAB Call
Power	$y = bx^m$	p = polyfit(log(x), log(y), 1)
Exponential	$y = be^{mx}$	p = polyfit(x, log(y), 1)
Exponential	$y = b10^{mx}$	p = polyfit(x, log10(y), 1)
Logarithmic	$y = m\ln(x) + b$	p = polyfit(log(x), y, 1)
Logarithmic	$y = m\log(x) + b$	p = polyfit(log10(x), y, 1)
Reciprocal	$y = \frac{1}{mx + b}$	p = polyfit(x, 1./y, 1)

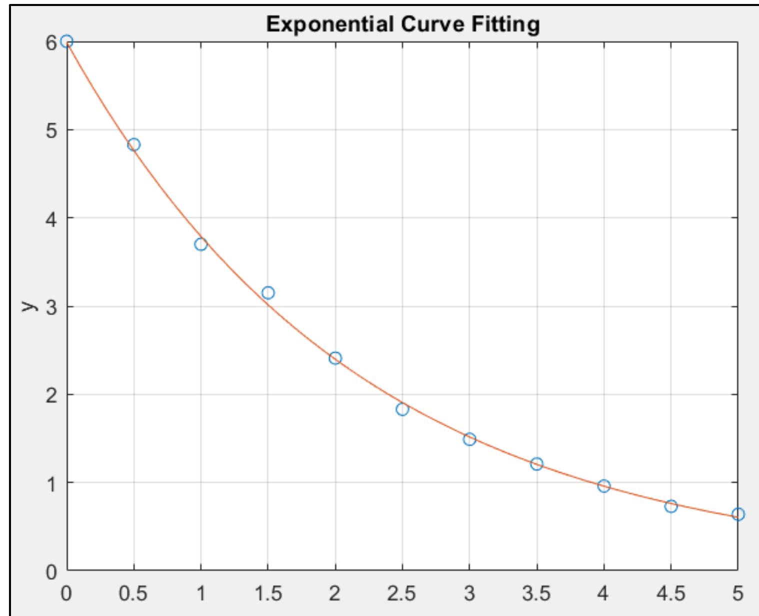
Example:

The following data is given for variables x and y:

x	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
y	6	4.83	3.7	3.15	2.41	1.83	1.49	1.21	0.96	0.73	0.64

Fit an exponential curve of the form $y = be^{mx}$ to this data. $[\ln(y) = mx + \ln(b)]$

```
% Given data
x = 0:0.5:5;
y = [6 4.83 3.7 3.15 2.41 1.83 1.49 1.21 0.96 0.73 0.64];

% Curve fitting using linearization
p = polyfit(x, log(y), 1);

% Extract constants
m = p(1);
b = exp(p(2));

% Generate smooth fitted curve
xm = 0:0.1:5;
ym = b * exp(m * xm);

% Plot data and fitted curve
plot(x, y, 'o', xm, ym);
xlabel('x');
ylabel('y');
title('Exponential Curve Fitting');
grid on;
```



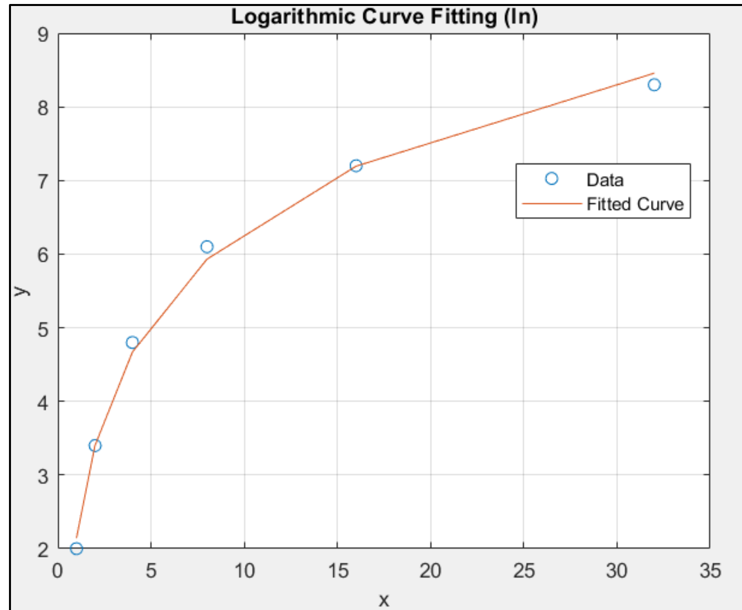
Example:

The following data is given for $x$and y:

x	1	2	4	8	16	32
y	2.0	3.4	4.8	6.1	7.2	8.3

Fit a logarithmic curve of the form $y = m\ln(x) + b$ to the given data.

```
x = [1 2 4 8 16 32];
y = [2.0 3.4 4.8 6.1 7.2 8.3];

p = polyfit(log(x), y, 1);

m = p(1);
b = p(2);
y_fit = m*log(x) + b;

plot(x, y, 'o', x, y_fit, '-');
xlabel('x');
ylabel('y');
title('Logarithmic Curve Fitting (ln)');
legend('Data','Fitted Curve');
grid on;
```



Curve Fitting App

The Curve Fitting App (also known as cftool) is a graphical user interface provided by MATLAB to analyze experimental or observed data and fit mathematical models to it. The app allows users to visually explore data, apply different curve fitting techniques, evaluate goodness of fit, and perform interpolation or extrapolation without writing extensive code.

The Curve Fitting App is part of the Curve Fitting Toolbox.

Procedure for 2DCurve Fitting in MATLAB

Step 1: Load the Data into the MATLAB Workspace

Data vectors can be entered directly in the Command Window by defining x and y, or imported from an Excel file using the Import Data option. After loading, verify that the variables appear in the workspace.

Step 2: Open the Curve Fitting App

Open the Curve Fitting application via Apps then clicks “Curve Fitting” or by typing cftool in the Command Window. The Curve Fitting window will appear.

Step 3: Import the Data into the App

Click the Data button and assign the variables by selecting x as the X data and y as the Y data. Leave the Z data empty for a two-dimensional fitting problem. Keep Weights set to (none) so that all data points are treated equally during fitting. (Weights are only required when some points are more reliable or more important than others.)

Step 4: Select the Fit Type

From the Fit Type menu, choose an appropriate model such as Polynomial (Linear, Quadratic, or Cubic), Exponential, Interpolant, or Custom Equation. The model should be selected based on the trend of the data rather than randomly.

Step 5: Set the Fit Options

Enable Auto Fit to perform the fitting automatically. If required, enable Center and Scale X data to improve numerical stability, especially for higher-order polynomial fits.

Step 6: Perform the Curve Fitting

Click Fit to generate the fitted curve. The fitted line will appear on the same plot along with the original data points.

Step 7: Analyze the Results

Observe the results panel to obtain the fitted equation, coefficient values, and goodness-of-fit statistics. Compare parameters such as SSE, R-square, Adjusted R-square, and RMSE. Smaller error values and an R-square closer to 1 generally indicate a better fit.

Step 8: Perform Residual Analysis

From the menu, select View- Residuals- Line Plot. Examine the residuals. A random scatter around zero indicates a good fit, while visible patterns or trends suggest the chosen model may be inappropriate.

Step 9: Compare Different Models

Use New Fit to try alternative models. Compare their error values and goodness-of-fit measures. Select the model that provides satisfactory accuracy instead of choosing an unnecessarily complex one.

Step 10: Saving the Fitted Curve

After completing the curve fitting in the Curve Fitting App, work can be saved directly from the app. In the Curve Fitting window, go to File – Print to Figure. Again go to File-Save As. Choose the desired file type (fig, .png, .jpg, or .pdf). Finally, select the location and click Save.

Practice Problem

1. A small plant grows as follows over 6 weeks:

Week	1	2	3	4	5	6
Height (cm)	3	5	8	12	18	25

Use MATLAB to estimate the plant's height at week 3.5 and week 5.5 using linear interpolation. Plot the original points with circles.

2. Plot the polynomial $y = -0.001x^4 + 0.051x^3 - 0.76x^2 + 3.8x - 1.4$ in the domain $1 \le x \le 14$. First create a vector for x, next use the polyval function to calculate y, and then use the plot function.

3. The number of bacteria N_B measured at different times t is given in the following table. Determine an exponential function in the form $N_B = Ne^{\alpha t}$ that best fits the data. Use the equation to estimate the number of bacteria after 60 min. Make a plot of the points and the equation.

t (min)	10	20	30	40	50
N_B	15,000	215,000	335,000	480,000	770,000

4. Find the curve of best fit of type.$y = ae^x$ for following data:

x	1	5	7	9	12
y	10	15	12	15	21

5. An investigator has reported the data tabulated below for an experiment to determine the growth rate of bacteria k (per d), as a function of oxygen concentration c (mg/L). Find which degree of polynomial is the best fit for the given data using MATLAB Curve Fitting Tool.

c (mg/L)	0.5	0.8	1.5	2.5	4
k (per d)	1.1	2.4	5.3	7.6	8.9