UNIT-III

Topics:-

Procedures and Functions:-

Arguments and return values, M-files, Formatted console input-output, String handling.

Control Statements:-

If, Else, Else-if, Repetition statements: While, for loop.

Procedures and Functions

1. Procedures (Scripts):-

A **procedure** in MATLAB usually refers to a **script file**. It's a sequence of MATLAB commands saved in an `.m` file.

Characteristics:

- No input or output arguments.
- Operates on data in the **base workspace**.
- Useful for automating repetitive tasks.
- All variables created in a script are accessible in the base workspace.
- Files with `.m` extension that execute a sequence of MATLAB commands.

Example:

```matlab
% file: calculateArea.m
r = 5;
area = pi * r^2;
disp(['Area of circle: ', num2str(area)]);
```

Run using:

```matlab
>> calculateArea
```

2. Functions

A **function** in MATLAB is a reusable block of code defined using the `function` keyword. It can take input arguments and return outputs.

Syntax:

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

Characteristics:

- Variables inside functions are **local** to the function (workspace isolation).
- Accepts inputs and returns outputs explicitly.
- Defined in separate `.m` files or nested within scripts/functions.

Types of Functions

- **Primary function**: The first function in an `.m` file.
- **Subfunction**: Additional functions in the same file as the primary function, not accessible from outside.
- **Nested function**: Defined within another function; can share workspace with the parent.
- **Anonymous function**: A one-line function with no name.

Example – Primary Function:

```matlab
function area = circleArea(radius)
    area = pi * radius^2;
end
```

Example – Anonymous Function:

```matlab
square = @(x) x^2;
result = square(5); % returns 25
```

Benefits of Using Functions

- **Modularity**: Breaks a large problem into smaller, manageable parts.
- **Reusability**: Functions can be reused across programs.
- **Maintainability**: Easier to debug and update.
- **Encapsulation**: Variables are local, avoiding accidental overwrites.

Feature	Scripts (Procedures)	Functions
File Extension	`.m`	`.m`
Input/Output	No arguments	Input & output arguments
Workspace	Shares workspace	Has its own local workspace
Usage	Automatically runs when called	Must be called with arguments

Arguments and Return Values in MATLAB:-

1. Input Arguments:-

Input arguments are the values or variables **passed to a function** when it is called. These inputs help the function perform specific tasks based on the data provided.

Syntax:

```matlab
function output = functionName(input1, input2, ...)
```

Example:

```matlab
function area = rectangleArea(length, width)
    area = length * width;
end
```

Call the function:

```matlab
>> a = rectangleArea(5, 10)
```

Function Arguments (Inputs)

- Functions can accept **input arguments** (parameters) to perform computations.
- Arguments are passed when the function is called.
- MATLAB functions can have:
  - **No input arguments**
  - **Fixed number of arguments**
  - **Variable number of arguments** (using `varargin`)

2. Return Values (Output Arguments)

Return values are the **results produced by the function** and sent back to the calling environment.

Call it as:

```matlab
>> [s, p] = calculateValues(4, 5)
```

3. Functions with No Inputs or Outputs

No Input, One Output:

```matlab
function r = giveRandom()
    r = rand();
end
```

No Input, No Output (used like a procedure):

```matlab
function greet()
    disp('Hello, MATLAB user!');
end
```

No Input, No Output (used like a procedure):

```matlab
function greet()
    disp('Hello, MATLAB user!');
end
```

4. Optional and Variable Number of Arguments

Variable Inputs (`varargin`) and Outputs (`varargout`):

Use when the number of inputs or outputs is not fixed. Stores inputs in a **cell array**.

Example:

```matlab
function result = addNumbers(varargin)
    result = 0;
    for i = 1:nargin
        result = result + varargin{i};
    end
end
```

Call:

```matlab
>> addNumbers(1, 2, 3, 4) % returns 10
```

Concept	Syntax Example	Description
Single Input	`function y = f(x)`	Takes one input `x`.
Multiple Inputs	`function z = g(x, y)`	Takes two inputs `x` and `y`.
Default Arguments	`if nargin < 2, n = 2; end`	Sets default if input missing.
Single Output	`function y = f(x)`	Returns one value.
Multiple Outputs	`function [a, b] = f(x)`	Returns two values `a` and `b`.
Ignore Output	`[~, b] = f(x)`	Skips the first return value.
Variable Inputs (`varargin`)	`function f(varargin)`	Accepts any number of inputs.

Concept	Syntax Example	Description
Variable Outputs (`varargout`)	`function [varargout] = f(x)`	Returns dynamic outputs.
Return Structure	`function s = f(x)`	Returns structured data.

M-Files in MATLAB

M-files are MATLAB script or function files saved with a `.m` extension. They allow you to:

- Store reusable code
- Organize complex programs
- Create custom functions

Types of M-Files

There are **two main types**:

1. **Script M-files**
2. **Function M-files**

A) Script M-Files

- Contain a sequence of MATLAB commands
- No input/output arguments
- Share workspace with the caller
- Used for automation and batch processing

Example (`myScript.m`):

```matlab
a = 5;
b = 10;
sum = a + b;
disp(sum);
```

You run it by typing:

```matlab
myScript
```

(B) Function M-Files

- Contain function definitions
- Have input/output arguments
- Have local workspace (variables don't persist)
- Must match filename with function name

Example (`addNumbers.m`):

```matlab
function result = addNumbers(a, b)
    result = a + b;
end
```

Use it like this:

```matlab
x = addNumbers(3, 7);
disp(x);  % Output: 10
```

2. Creating and Managing M-Files

Creating New M-Files

1. In MATLAB Editor:
    - Click **New Script** or **New Function**
    - Or use `edit filename.m`
2. Using Command Window:

```
>> edit myscript.m
```

Formatted Console Input/Output

Formatted input-output in MATLAB refers to the controlled way of displaying data to the console (output) or reading data from user input (input) with specific formatting rules. This allows you to precisely control how numbers, text, and other data types are presented or interpreted.

1. Formatted Console Output

Formatted output in MATLAB is primarily done using the `fprintf` function, which writes formatted data to the command window or a file according to specified format strings.

- **`fprintf` – *Formatted Print to Console***

Definition:
`fprintf` is a MATLAB function used to **display text** in the Command Window with formatting, similar to `printf` in C.

Syntax:

```
fprintf('format_string', variables);
```

Example:

```
name = 'Alice';
age = 25;
fprintf('Name: %s\nAge: %d\n', name, age);
```

Explanation:

- `%s` is a **format specifier** for a string.
- `%d` is for an **integer**.
- `\n` means **new line**.

Common Format Specifiers

- `%d` - **Integer**
- `%f` - **Fixed-point notation**
- `%e` - **Exponential notation**
- `%g` - **Compact format (automatically selects %f or %e)**
- `%s` - **String**
- `%c` - **Single character**

- **`disp` – *Display Simple Output***

Definition:
`disp` is used for displaying **simple messages or values** without formatting.

Example:

```
disp('Hello, World!');
```

Use Case: Good for quick messages without the need for formatting.

2. Console Input

Formatted input in MATLAB is primarily done using the `textscan` function for reading formatted data from files or strings, or `input` for interactive console input.

- ***`input` – Take Input from User***

Definition:
`input` is used to **read data from the user** during program execution.

Syntax:

```
x = input('Prompt message: ');
```

Example:

```
age = input('Enter your age: ');
```

To accept a string, use:

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

`'s'` tells MATLAB to treat the input as a **string**.

Format Specifiers (Used in `fprintf`)

Specifier	Meaning	Example
`%d`	Integer	`Age: %d`
`%f`	Floating-point (default 6 decimals)	`GPA: %f`
`%.2f`	Floating-point with 2 decimals	`GPA: %.2f`
`%s`	String	`Name: %s`
`\n`	New line	
`\t`	Tab space	

Full Example with Explanation

```
name = input('Enter your name: ', 's');  % Takes string input
age = input('Enter your age: ');         % Takes integer input
gpa = input('Enter your GPA: ');         % Takes floating-point input

fprintf('\nStudent Information:\n');     % Prints header
fprintf('Name: %s\n', name);             % Prints string
fprintf('Age: %d years\n', age);         % Prints integer
fprintf('GPA: %.2f\n', gpa);             % Prints float with 2 decimal places
```

Formatting Tricks

- `\n` – newline
- `\t` – tab
- `%.2f` – float with 2 decimal places
- `%d` – integer
- `%s` – string

Key Features

1. **Precision Control**: You can specify exactly how many decimal places to display.
2. **Alignment**: Numbers and text can be left or right aligned in fields.
3. **Mixed Data Types**: You can combine numbers, strings, and other types in one output statement.
4. **Special Characters**: Format strings can include newlines (`\n`), tabs (`\t`), etc.

When to Use

- When you need precise control over how data is displayed
- When creating reports or tables in the command window
- When reading data files with specific formats
- When you need user input in a specific format

Formatted I/O is particularly useful for creating professional-looking output and for processing data files with fixed formats.

String Handling

String handling in MATLAB refers to the manipulation, analysis, and processing of text data (strings) using built-in functions and operators. MATLAB treats strings as arrays of characters or as string arrays (introduced in R2016b), allowing various operations like concatenation, searching, comparison, and modification.

Types of Strings

1. Character Arrays (Legacy Strings):-

- Represented as `'text'` (single quotes)
- Each character is an element in an array
- Example:

```
str = 'Hello MATLAB';
```

2. String Arrays (Modern Approach, R2016b+)

- Represented as `"text"` (double quotes)
- Treated as a single object (not an array of characters)
- Supports vectorized operations
- Example:

```
str = "Hello MATLAB";
```

Key String Operations

1. Creating Strings

Definition: Creating a sequence of characters stored in a variable.

- **Character Array:** Uses single quotes `' '`.
- **String Array (Newer):** Uses double quotes `" "`.

Example:

```
charStr = 'Hello';    % Character array
stringStr = "World";  % String type
```

2. Concatenation

Definition: Joining two or more strings together.

Example:

```
% Character array
fullCharStr = ['Hello', ' ', 'World'];  % Result: 'Hello World'

% String array
fullStringStr = "Hello" + " " + "World";  % Result: "Hello World"
```

3. Comparing Strings

Definition: Checking if two strings are equal or not.

Functions:

- `strcmp()` — for character arrays
- `==` or `~=` — for string arrays

Example:

```
strcmp('abc', 'abc')     % true
"abc" == "abc"           % true
"abc" ~= "xyz"           % true
```

4. Finding Substrings

Definition: Searching for the presence or position of a substring in a string.

Functions:

- `contains()` — returns true if the substring exists
- `strfind()` — returns the position index of the substring

Example:

```
contains("Hello World", "World")  % true
strfind('Hello World', 'World')   % 7
```

5. Extracting Substrings

Definition: Getting a specific portion of a string.

Function:

- **`extractBetween()`**

Example:

```
str = "Programming";
extractBetween(str, 1, 6)   % "Progra"
```

6. Changing Case

Definition: Converting all letters in a string to uppercase or lowercase.

Functions:

- `upper()` — Converts to uppercase
- `lower()` — Converts to lowercase

Example:

```
matlab

upper("matlab")   % "MATLAB"
lower("MATLAB")   % "matlab"
```

7. Replacing Substrings

Definition: Replaces a specific part of a string with another string.

Functions:

- **`replace()`** — for string arrays
- **`strrep()`** — for character arrays

Example:

```
replace("Hello World", "World", "MATLAB")   % "Hello MATLAB"
strrep('Hello World', 'World', 'MATLAB')    % 'Hello MATLAB'
```

8. Splitting Strings

Definition: Breaking a string into parts based on a delimiter.

Functions:

- `split()` — for string arrays
- `strsplit()` — for character arrays

Example:

```
split("one,two,three", ",")     % ["one", "two", "three"]
strsplit('one two three')       % {'one', 'two', 'three'}
```

9. Joining Strings

Definition: Combining elements of a string array into one string with a separator.

Function:

- **`join()`**

Example:

```
join(["apple", "banana", "cherry"], ", ")  % "apple, banana, cherry"
```

Convert Between Strings and Numbers

Number to String:

1. `num2str()`

Definition: Converts a number (or array of numbers) to a character array (text format).

Example:

```
str1 = num2str(123.45);       % str1 = '123.45'
str2 = num2str([1 2 3]);      % str2 = '1  2  3'
```

2. `string()`

Definition: Converts a number to a **string object** (not character array).

Example:

```
str3 = string(456.78);        % str3 = "456.78"
```

String to Number:

1. `str2double()`

Definition: Converts a string to a double-precision number. Returns `NaN` if the string is not a valid number.

Example:

```
num = str2double("123.45");   % num = 123.45
num2 = str2double("abc");     % num2 = NaN
```

2. `str2num()`

Definition: Converts a character array or string to a number (or array of numbers). Slower than `str2double`, but can evaluate MATLAB expressions.

Example:

```
x = str2num('3.14')           % x = 3.14
y = str2num('1 2 3')          % y = [1 2 3]
```

Note: `str2num()` uses `eval()` internally, so avoid it with user input for security reasons.

Advanced String Handling

1. `regexp` – Regular Expressions

Definition: Finds patterns in strings using regular expressions (regex), which are powerful pattern-matching tools.

Uses:

- Find specific patterns
- Extract matching parts
- Replace complex string patterns

Example:

```
str = "email: user123@example.com";
match = regexp(str, '\w+@\w+\.\w+', 'match');
% Output: "user123@example.com"
```

2. `regexprep` – Regex Replacement

Definition: Replaces parts of a string that match a regex pattern.

Example:

```
str = "The price is $100";
newStr = regexprep(str, '\$\d+', '$200');
% Output: "The price is $200"
```

Control Statements

Control statements **direct the flow of execution** in a program. They are essential in making decisions and repeating certain parts of code.

1. Conditional Statements

These statements **allow decision-making** in programs based on specific conditions. If a condition is true, one block of code executes; otherwise, a different block may execute.

◆ *`if` Statement*

Definition:

The `if` statement is used to **execute a block of code only if a specified condition is true**.

Syntax:

```
if condition
    % code to execute
end
```

Example:

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

In this example, the message is displayed only if `x > 5`.

◆ `if-else` Statement

Definition:
An `if-else` statement is used to **execute one block of code if the condition is true and another block if the condition is false**.

Syntax:

```
if condition
    % code if condition is true
else
    % code if condition is false
end
```

Example:

```
x = 3;
if x > 5
    disp('x is greater than 5');
else
    disp('x is 5 or less');
end
```

🔹 `if-elseif-else` **Statement**

Definition:
The `if-elseif-else` structure is used when there are **multiple conditions to check**.

Syntax:

```
if condition1
    % block 1
elseif condition2
    % block 2
else
    % default block
end
```

Example:

```
x = 7;
if x < 5
    disp('Less than 5');
elseif x == 5
    disp('Equal to 5');
else
    disp('Greater than 5');
end
```

This allows the program to select one of many code paths based on the conditions.

2. Repetition Statements (Loops)

Loops are used to **repeat a set of instructions** multiple times, which helps in reducing redundancy in code.

◆ **`while`** Loop

Definition:
A `while` loop is used to **repeatedly execute a block of code as long as the given condition is true**.

Syntax:

```
while condition
    % code to execute
end
```

Example:

```
x = 1;
while x <= 5
    disp(x);
    x = x + 1;
end
```

In this loop, `x` is printed repeatedly until it becomes greater than 5.

🔹 `for` Loop

Definition:
A `for` loop is used to **repeat a block of code a specific number of times**, usually with a counter variable.

Syntax:

```
for variable = start:step:end
    % code to execute
end
```

Example:

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

This loop prints values from 1 to 5. The variable `i` increases automatically in each iteration.

With step:

```
for i = 1:2:9
    disp(i);  % Output: 1, 3, 5, 7, 9
end
```

3. Control Transfer Statements

Used to control the flow inside loops.

1. **`break` statement**

   Exits the loop prematurely.

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

2. **`continue` statement**

   Skips the current iteration and continues with the next.

```
for i = 1:10
    if mod(i,2) == 0
        continue;
    end
    disp(i)
end
```

4. Key Concepts

1. **Condition Evaluation**: MATLAB uses logical `true` (1) or `false` (0)
2. **Nested Structures**: Control statements can contain other control statements
3. **Vectorization**: Often preferable to loops for performance
4. **Indentation**: Improves readability (not required but recommended)