CHAPTER 1

NUMERICAL ALGORITHMS AND ERRORS

SUPPLEMENT No. 1d:

MATLAB TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1d: The MatLab Tutorial

LEARNING OBJECTIVES

The objective of this Matlab programming tutorial is to
- present a short summary of the basics of Matlab;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs;
- explain how to prepare functions or subprograms.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing the methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran, Visual Basic, Python, Matlab®, and Mathematica®, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running Matlab programming language. It is not intended to be a "complete language reference document." The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the Matlab programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the Matlab programming language.

1 MATLAB BASIC FEATURES

Matlab is an interpreted language allowing interactive programming. It has an interactive terminal that prompts so-called the command prompt (≫). The command prompt receives instructions from the user. The user can command it by typing the command in and hitting the *Enter* button. Then, Matlab evaluates the command entered and displays the results.

In general, Matlab, which is an interpreted environment, can function like a calculator. It will execute the entered command right away, and the result is returned as `ans =`*result*. In this way, in an interactive session, calculations can be performed step by step. The final answer can then be recorded or saved. In this process, a number of *variables*, named data, may be used as follows:

```
>>  1 + 3
 ans = 4
>> x = 5
 x = 5
>> y = 11
 y = 11
>> 5 * x + 2 * y
 ans = 47
```

Table 1.1: Special MATLAB characters and commands for managing a work session.

Command	Purpose	Character	Meaning
`clc`	Clear command window	`=`	Assignment
`clear`	Remove variables from memory	`()`	Prioritize operations
`exist`	Check for existence of file or variable	`[]`	Construct array
`global`	Declare variables to be global	`{}`	Cell arrays
`help`	Searche for a help topic	`:`	Specify range of array elements
`lookfor`	Searche help entries for a keyword	`;`	Column element separator
`quit`	Stop Matlab	`,`	Row element separator
`who`	List current variables	`%`	Insert comment after it
`whos`	List current variables (long display)	`...`	Continue to next line

MATLAB has commands for managing an active session. A short list of commands is presented in **Table 1.1**. For instance, the `help` and `lookfor` commands are designed to provide information about a known command (e.g. `help zeros`) or to perform a task (e.g. `lookfor roots`). Special characters, which are also listed in **Table 1.1**, have various meanings and uses in MATLAB.

1.1 VARIABLES

A MATLAB variable is an identifier name that is assigned to a value while that it remains in memory. This identifier provides a way to reference a value in memory so that it can be retrieved from memory, operated with other data, and stored back into memory.

A variable identifier name (max. 63 character-long) must begin with a letter, which can be followed by any combination of letters, digits, and underscores (e.g., `varA, varB_12, _myFlag`, etc). MATLAB keywords (such as `break`, `case`, `catch`, `elseif`, `global`, etc.) are reserved for specific tasks and cannot be used as identifiers. A properly defined variable can be accessed and used numerous times anywhere and anytime.

Variables must be initialized before they can be used. When an expression that is not assigned to a variable is processed, MATLAB assigns it to a variable named `ans`, which can be accessed and used later if `ans` is not overwritten with another unassigned expression.

MATLAB variable names are case-sensitive. When defining and referencing variables, use the exact same casing each time. Also, it is good programming practice to declare global variables, usually typed in uppercase letters, at the beginning of any function.

Following examples illustrate assignment to a local variable, by value or by an expression. The symbol % is used for indicating a full or partially reserved comment line.

```
>> (2^8-1)/(2^4-1)                % expression not assigned a variable
 ans = 17                         % result is stored on ans
>> ans*2^3                        % result can be accessed using ans
 ans = 136                        % ans is replaced with 136
```

Every MATLAB subprogram (i.e., function) also has variables of its own, called the *local variables*. These variables are separate from those of the base workspace or other function subprograms, and their values are accessible within the subprograms, i.e., *not shareable with other functions*. On the other hand, variables called *global variables* can be shared. A variable must first be declared as `global` before it can be shared with a function. In the following example, `MAXSIZE` and `MAX_ITERATIONS` are declared as `global`, along

Table 1.2: Special variables and constants.

Command	Purpose
`ans`	Temporary variable containing the most recent answer
`eps`	Gives the accuracy of floating point precision, ε
`i,j`	The imaginary unit, ß $= sqrt{-}1$, $j = sqrt{-}1$
`inf`	Infinity, ∞
`pi`	π
`e`	$e = 2.71828$
`NaN`	**N**ot **a** **N**umber.

with the specified global values. The global variables in a large code can be inquired about by the `whos` statement.

```
>> global MAXSIZE MAX_ITERATIONS
MAXSIZE = 99; MAX_ITERATIONS = 999;

>> whos global    % inquire available global variables with 'global'
Global variables:
  Attr   Name               Size                     Bytes  Class
  ====   ====               ====                     =====  =====
     g   MAXSIZE            1x1                          8  double
     g   MAX_ITERATIONS     1x1                          8  double
Total is 2 elements using 16 bytes
```

MATLAB stores variables in a part of memory called an *workspace*. A variable need not be *type-declared* in M-files (with the exception of `global` variables). Before assigning a variable to another one, make sure that the variable on the right-hand side of an assignment statement has a value.

Table 1.2 provides a number of commonly used special variables and constants (π, i, ∞, ε, ...) that are defined in MATLAB by default. In the following examples, the value of `pi` is available by default and is used in arithmetic operations.

```
>> pi             % pi is defined constant by default
ans = 3.1416
>> e              % e is defined constant by default
ans = 2.7183
>> (2+3i)*(2-3i)  % arithmetic operation with imaginary 'i'
ans = 13
>> (2+3*i)/3      % 3i and 3*i yield the same result
ans =  0.6667 + 1.0000i
>> R=5            % set a numerical value for the radius
R = 5
>> V=4*pi*R^3/3   % calculate volume of sphere of radius R, pi is default
V = 523.60
```

Table 1.3: Numeric output format options.

Style	Example
`short` (default option)	`3.1416`
`long`	`3.141592653589793`
`shortE`	`3.1416e+00`
`longE`	`3.141592653589793e+00`
`rat`	`355/113`

1.2 CONTROLLING DISPLAYS

MATLAB uses double-precision floating-point arithmetic by default, which conforms to the IEEE 754 standard. This provides a precision of approximately 15 to 17 significant decimal digits. However, by default, MATLAB prints out only 4 decimals of the calculations, e. g., `22/7` gives `3.1429`. The display format of a variable can be set in the *Command Window* or *Editor* using the `format` function. Some of the format options available in MATLAB are presented in **Table 1.3**.

The following examples illustrate the use of the format function:

```
>> 22/7           % format short (default setup)
 ans = 3.1429
>> format long    % format is set to long
>> 22/7
 ans = 3.142857142857143
>> format longe   % format is set to longe
>> 22/7
 ans = 3.142857142857143e+00
>> format rat     % format is set to ratio of small integers
>> pi
 ans = 355/113
```

1.3 DATA TYPES

MATLAB provides different types of data (integer, floating point, character, string, logical, etc.) that one can work with. One can also create his own data types using MATLAB classes. Two of the MATLAB data types, structures and cell arrays, provide a way to store dissimilar types of data in the same array.

Integers: MATLAB has four *signed* and four *unsigned* integer data types. Signed integers allow one to work with negative and positive integers. The unsigned types allow one to work with a wider range of numbers, i.e., the 64-bit unsigned integer range is $0 \leqslant n \leqslant 2^{64} - 1$. The signed 32-bit integer type allows the range of numbers in $-2^{31} \leqslant n \leqslant 2^{31} - 1$.

Floating-Point Numbers: MATLAB represents floating-point numbers in either *single* or *double-precision* format. The default type of real numbers is double precision. The `float` and `double` are functions used to convert integer data to specific floating point numbers.

Characters and Strings: In MATLAB, the term string refers to an array of characters. You can use `char` to hold an $m \times n$ array of strings as long as each string in the array has the same length. A string is created as a vector whose components are the numeric codes for the characters. The actual characters displayed depend on the character set encoding for a given font.

Complex Numbers: Complex numbers are defined with two parts: a real and an imaginary part. In MATLAB, the basic imaginary unit is equal to the square root of -1, and it is represented by either i or j. For example,

```
>> x=2+3i          % both 3i and 3*i uses are legal
 x =  2 + 3i
>> y=(3+2j)*i      % i and j do not cause conflit in the same expression
 y = -2 + 3i
>> x+y
 ans =  5 + 5i
>> x*y
 ans =   0 + 13i
```

Logical Types: The logical data type represents a logical *true* (i.e., 1) or *false* (i.e., 0) state using numbers 0 and 1. Some MATLAB functions and operators return *true* or *false* to indicate whether a certain condition was met or not. For example, the statement (5 * 10) > 40 returns a logical true value.

```
>> 10>9
 ans = 1          % indicating the logical expression gives 'true'
>> (10>9)<(5<=25)
 ans = 0          % indicating the logical expression gives 'false'
```

MATLAB also allows many more data types, such as Dates and Time, Tables, Structures, Cell arrays, Dictionaries and so on. For a complete list, visit www.mathworks.com.

2 ARITHMETIC, RELATIONAL, AND LOGICAL OPERATORS

In **Table 1.4**, the arithmetic, relational, and logical operators that are available in MATLAB are summarized. Arithmetic Operators are `+, -, *, /, \, .*, ./, .\, .^`, and they are used to perform numeric computations such as adding, subtracting, multiplying, and dividing of two scalar or array variables.

Expressions involving the arithmetic operators often involve the assignment operator `=`. The general form of an assignment statement is

```
Variable-name = expression  (or a value)
```

In MATLAB, an *expression* on the right-hand side of the '=' sign is evaluated first, and its evaluated value is placed at the allocated memory location of the *variable* on the left-hand side. Any recently computed value of *variable* replaces its previous value. An *assignment* denoted by ← (a left-arrow) in pseudocode notation is replaced with a '=' sign. Variables can be initialized by specified values in the same way, i.e., `pi = 3.1416`, `X=0`, etc.

In arrays, arithmetic operators work on corresponding elements of the same size arrays. If one operand is a scalar, MATLAB applies the scalar to every element of the other operand. These kinds of operations are referred to as the *array arithmetic operations*.

When dealing with matrices or arrays, MATLAB offers two kinds of arithmetic operations: (i) *matrix arithmetic operations* (*see* Section 5.8) and (ii) *array arithmetic operations* (element-by-element ops).

Relational operators (`<, >, <=, >=, ==, ~=`) are used to compare corresponding scalars or elements of arrays with equal dimensions, returning a logical result (true or false). When applied to arrays, the relational operators always operate on an element-by-element basis and return a logical array. Effective use of relational operators allows controlling the flow and making decisions in MATLAB programs.

Table 1.4: Arithmetic, equality/relational, logical and assignment operators in C.

Operator	Description	Example
ARITHMETIC OPERATORS		
$\pm$	Addition/subtractions (scalar/element-by-element)	`a` $\pm$ `b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
$\pm$	Element-by-element and matrix operations	`A` $\pm$ `B`
`.*`	Element-by-element multiplication	`A .* B`
`./`	Element-by-element division	`A ./ B`
`.^`	Element-by-element exponentiation	`A.^ n`
RELATIONAL OPERATORS		
`==`	compares the operands to determine equality	`a == b`
$\sim$`=`	compares the operands to determine unequality	`a` $\sim$`= b`
`>`	determines if first operand greater	`a > b`
`<`	determines if first operand smaller	`a > b`
`<=`	determines if first operand smaller than or equal to	`a > b`
`>=`	determines if first operand greater and equal to	`a > b`
LOGICAL OPERATORS		
`&` or `and`(ℓ_1, ℓ_2)	Logical operator, **and**	`a&b` or `and(a,b)`
`|` or `or`(ℓ_1, ℓ_2)	Logical operator, **or**	`a|b` or `or(a,b)`
$\sim \ell$ or `not`(ℓ)	Logical operator, **not**	$\sim$ `a` or `not(a)`

Note: `a, b` and `A, B` denote scalars and arrays, respectively.

Logical operators, `&` (`and`), `|` (`or`), and $\sim$ (`not`), and logical functions (`any`, `all`, `find`, `isequal`, and `xor`) are essential for performing logical operations, manipulating logical arrays, and controlling program flow. *Logical_expression* can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result.

MATLAB offers three types of logical operations: element-wise (operate on corresponding elements of logical arrays), bit-wise (operate on corresponding bits of integer values or arrays), and short-circuit (operate on scalar, logical expressions).

Branching structures are controlled by *logical variables* and *logical operations*. Logical operators evaluate relational expressions to either 1 (`True`) or 0 (`False`). Logical operators are typically used with Boolean operands. The logical 'and' operator (&) and the logical 'or' operator (|) are both binary in nature (require two operands). The logical 'not' operator ( ) negates the value of a Boolean operand, and it is a unary operator. The `and` (`&`) operator has precedence over the `or` (`|`) operator. MATLAB evaluates logical expressions from left to right. However, in mixed logical statements, use parentheses to explicitly specify the intended precedence of statements containing combinations of `&` and `|`. For instance, instead of `a|b|b&c` use `(a|b)|(b&c)`.

3 INPUT/OUTPUT (I/O) STATEMENTS

MATLAB automatically displays the results of expressions as they are executed, such as when a value is assigned to a variable or when the name of a variable already defined is typed and the `Enter` key is pressed. However, if a semicolon (`;`) is typed at the end of a command line, no output is displayed.

Besides the foregoing feature, MATLAB has functions such as `disp` and `fprintf` designed to gener-

ate outputs that provide information, numerical data, plots, etc. The **disp** is used to display data in the command-line window without returning a value. It is often used for printing messages or showing the results of calculations. On the other hand, **fprintf** allows data to be displayed in a specified format in the command window or in a file, and it is more versatile than **disp** because it allows out to be formatted.

3.1 THE input FUNCTION

The input function is used to receive input from the user during the execution of a program.

```
variable = input(prompt)
```

where **prompt** denotes a string that specifies the message to display to the user, and **variable** denotes the variable to which the entered value is to be assigned.

This function will display the message expressed in **prompt** in the command window and wait for the user to input a number. The value entered is assigned to the variable (name) on the left-hand side. In the following code statements, the data for the two variables (**name** and **ssn**) is supplied and registered in the memory addresses upon entering values based on prompted messages.

```
name= input('Enter your name & last name: ');        % enter value=> name
ssn = input('Enter your social security number: '); % enter value=> ssn
```

3.2 THE disp FUNCTION

This command displays the variables without the name of the variables and without any text. The syntax for **disp** is given as

```
disp(var_name)
```

where **var_name** can be a string, numeric value, array, or any other MATLAB variable to be displayed. Every time the **disp** is executed, it displays the result in a new line.

To display a string along with variable values, functions sprintf or fprintf need to be used, but disp does not support direct concatenation of strings and variables. However, a variable can be converted to a string, and then disp can be used. A string and a number using a **disp** function can be displayed only after converting the numbers into strings using **num2str** function and placing the combinations in a string array.

Following examples illustrate how simple and straightforward it is to output information without formatting:

```
>> x=12; y=21; z=33   % prints out only the last variable
 z = 33
>> disp('x');disp(x)  % displays string and numerical variables on new lines
 x=
 12
>> disp(y);disp(z)    % displays y and z on new lines
 21
 33
>> format long        % output precision is set
>> pi
 ans = 3.141592653589793
>> disp(pi)           % disp obeys the precision set in workspace
 3.141592653589793
>> disp(['pi=',num2str(pi,5)]) % string and a number in the same line
```

```
pi=3.1416
>> name = 'Robert'; age = 25;
disp(['Name: ' name ', Age: ' num2str(age)])
```

In the last example above, the age is converted and then two strings are concatenated.

3.3 THE `fprintf` FUNCTION

The **`fprintf`** can be used to display formatted data on the window command line or to save the output to a file. The syntax for **`fprintf`** is as follows:

```
fprintf(formatspec, v1, v2, ..., vn)
```

where **`formatspec`** denotes the format specification, a string that specifies the format of the output, which may include text as well as format specifiers that define how to display the subsequent arguments. The **`v1, v2, ..., vn`** refer to one or more variables or values to be printed according to the format specified in **`formatspec`**.

The **`fprintf`** function allows a set of string or numerical outputs to be mixed and displayed on the same line with the desired format. This is accomplished with the format specifications that control the notation, alignment, significant digits, field width, and other aspects of output format. Most commonly used format characters, flags, etc. are presented in **Table 1.5**.

The most general form of a format specification statement can be expressed as follows:

```
%+w.ptc
```

where

% sign is required to mark the location of the output to be inserted into the string.

Flags (**`+,-,' ',0,`** etc.), which are *optional*, allow the user to left justify, print a sign character (+ or -), pad values with zeros, and so on.

Field width (**`w`**) is *optional*, and it specifies the number of spaces reserved for the entire value to be printed. In the width specification, every single character (period, + or - signs, letters, numbers, etc.) occupies one space.

Precision (**`p`**) is also *optional* and specifies the number of digits to the right of the decimal point when %f or %e format specifiers are used. It is optional and used with a period in front of it. When using %g, the precision **`p`** denotes the total number of significant digits, including the left of the decimal.

Subtype (**`t`**) field, also *optional*, is a single alphabetic character. Without the subtype field, the %o, %x, and %u conversion characters treat input data as integers. To treat input data as floating-point values instead and convert them to octal, decimal, or hexadecimal representations, use one of the following subtype specifiers.

Conversion character (**`c`**) is mandatory and indicates the type of the output data (**`d`** integer, floating points (**`e, f, g`**,..), characters (**`c, s`**,...). A conversion character suitable for the output data type should be used.

Some formatting examples are given below:

```
>> x=12.3456; y=1.234e2;      % does not print values
>> fprintf('x= %f y=%e',x,y)
 x= 12.345600 y=1.234000e+02  % uses default settings for f and e formats
>> fprintf('x= %10.2f y=%12.2e',x,y)
 x=      12.35 y=    1.23e+02
```

Table 1.5: Description of conversion and escape characters, types, flags, field width, and precision.

CONVERSION CHARACTERS	
`%d`	Integer (signed) numbers in base 10
`%f`	Fixed-point notation
`%e`	(or `%E`) Scientific notation of floating points
`%g`	(or `%G`) The more compact of %e or %f, with no trailing zeros
`%c`	Single character

FLAGS	
−	Minus sign left-justifies the number in its specified field (i.e., `%-6.3d`)
+	Minus sign displays the sign (+ or −) of the number (i.e., `%+6.3d`
Space	Blank space inserts a space before the value (i.e., `% 6.3d`)
`0`	Zero pads with zeros rather than spaces (i.e., `%06.3d`)

OUTPUT FIELD AND PRECISION	
`w`	Fixed width, specifies min. number of digits to be printed (e.g., %8f)
`p`	Precision, number of digits to be printed to after decimal point (e.g. `%8.3f`)

ESCAPE CHARACTERS	
`\b`	backspace
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab

SUBTYPES	
`b`	If value is a double-precision floating-point number rather than unsigned integer.
`t`	If value is a single-precision floating-point number rather than unsigned integer.

```
>> fprintf('x= %+10.2f y=%+10.2e',x,y)
 x=     +12.35 y= +1.23e+02
>> fprintf('x= %+10.2f\ny=%+10.2e',x,y)  % \n creates a new line
 x=     +12.35
 y= +1.23e+02                % format specificati after \n is printed here
>> fprintf('x= %2.1f\ny=%+5.3e',x,y)
 x= 12.3
 y=+1.234e+02
>> m=123;k=12345;fprintf('M=%10dK=%3d',m,k)
 M=       123K=12345
```

Note that if `w` is larger than necessary, extra blank spaces are added in the output. By default, the output values are right-aligned, i.e., the padded blank spaces go in front (to the left of) the value printed (*see* lines 2, 7, 9, 15). If `w` is smaller than the field width required to print the value, MATLAB automatically increases the field width as necessary to fit the value. In lines 13 and 15, the fields are insufficient, but the compiler has taken the necessary measures to print out the numbers without compromising their value.

The `fprintf` is vectorized, which also enables printing vectors and matrices with compact expressions until all the elements are displayed. In the following examples, the arrays and matrices are printed without the need for loops.

Table 1.6: Format strings in MATLAB.

Format String	Explanation
`%12.4f`	use floating-point format (`%f`) to convert a numerical value to a string 12 characters wide with 4 digits after the decimal point.
`%15.5e`	use scientific notation format (`%e`) to convert numerical value to a string 15 characters wide with 5 digits after the decimal point. The 15 characters for the string include the `e+00` or `e-00` (or `e+000` or `e-000` on Windows).
`%8.6g`	use g-format (`%g`) to convert numerical value to a string 8 characters wide with 6 significant digits. The 8 characters for the string include the `e+00` or `e-00` if the number is too large or too small.

```
>> x=1:2:9; y=sqrt(x)      % create an array x in [1,9] with steps of 2
 y =                        % y = sqrt(x) is applied to all x resulting in
    1.0000   1.7321   2.2361   2.6458   3.0000
>> fprintf('y=%9.4f\n',y)  % applies format for all x and y's
 y=   1.0000
 y=   1.7321
 y=   2.2361
 y=   2.6458
 y=   3.0000
>> A = [3 2 -1; 2 -4 3; -1 8 2]
 A =                        % unformatted output
    3   2  -1
    2  -4   3
   -1   8   2
>> fprintf('%8.2f %8.2f %8.2f\n',A)   % applies to all rows
    3.00     2.00    -1.00
    2.00    -4.00     8.00
   -1.00     3.00     2.00
```

4 PROGRAM CONTROL OPERATIONS

In MATLAB, program control operations allow you to dictate the flow of execution within your program. These operations are viewed in four categories: Conditional control statements (`if` and `switch`), Loop control statements (`for` and `while`), Control flow commands (`continue`, `break`, and `return`), Error handling (`try-catch`).

4.1 CONDITIONAL CONTROL: `if-end` AND `if-else-end` CONSTRUCTION

In a program, the constructions in this group allow the direction of the process to be changed or to make decisions. The flow path (code blocks to be executed) is based on whether a <*condition*> (boolean expression) is `true` or `false`. MATLAB supports the following variants of `if` constructs:

```
if (condition)          % equivalent to If-Then construct in the pseudocode
   STATEMENTS           % if condition is true
end
```

```
if (condition)          % equivalent to If-Then-Else construct in the pseudocode
   STATEMENTS-t         % if condition is true
else
   STATEMENTS-f         % if condition is false
end
```

or

```
if (condition_1)
   STATEMENTs-1         % if condition_1 is true
elseif (condition_2)
   STATEMENTs-2         % if condition_2 is true
else
   STATEMENTs-3         % if condition_2 is false
end
```

One may chain multiple conditions using **elseif** to test multiple possibilities. This can be achieved by repeating the **elseif** statement. Also, nesting **if** statements within the block statements **if** constructs is allowed. Following are examples of **if** constructions:

```
if ( num > 0 )
   disp('num is a positive integer')
else
   disp('num is zero or a negative integer')
end
        ...
if (i < j && i < k)
   num = i                % gives the minimum of 3 numbers
elseif (j < i && j < k)
   num = j
else
   num = k
end
```

In MATLAB, the *condition* in any conditional construction does not require parentheses. However, they can be included to make complex conditions easier to read.

4.2 CONDITIONAL CONTROL: switch-case-otherwise STRUCTURE

The **switch** construct is used to execute different code blocks based on the value of a single variable or expression. It is often cleaner and more readable than using multiple **if-elseif** statements, especially when dealing with many conditions.

The **switch** construct uses **case** statements (may also include **otherwise**) to select from a set of options depending on the value of an **expression**. In this structure, *expression* is evaluated first. The flow is then directed to the case block (code statements), matching the corresponding value to the expression. Execution of a case block ends when the next **case** statement or the **otherwise** statement is reached. The **otherwise** block is *optional*; it is only executed if the value of the *expression* is not handled by any one of the cases.

The general syntax is shown below:

```
switch expression        % may be a scalar or a string.
    case value-1
        STATEMENTS-1     % execute if expression is value-1.
    case value-2
        STATEMENTS-2     % execute if expression is value-2.
      . . .
    otherwise
        STATEMENTS-o     % execute if expression does not match
                         % any other case.
end
```

Consider the following MATLAB code:

```
year=3
switch year
    case 1
        disp('Freshman')
    case 2
        disp('Sophomore')
    case 3
        disp('Junior')       % Output is Junior for year=3
    case 4
        disp('Senior')
    otherwise
        disp('Graduated')
end
```

This code uses **year** to execute **switch-case** construction. For the case **year=1**, **Freshman** is displayed; for **year=2**, **year=3**, and **year=4**, **Sophomore**, **Junior**, and **senior** are displayed, respectively. If **year** corresponds to none of the above, the message **Graduated** is displayed.

To execute multiple cases, the values can be listed on the same line: The **switch** can handle multiple conditions in a single **case** statement by enclosing the case expression in a cell array, i.e., **case {5-7}** or **case {1, 4, 8-11}**. Such a case is illustrated in the following code:

```
year=3
switch year
    case {1-4}
        disp('Undergrad stundent')
    case {5, 6}
        disp('Graduate Student')
    otherwise
        disp('Graduated')
end
```

Unlike some programming languages, the **switch** construct does not fall through to the next case. It executes the block for the matching case and exits **switch**.

4.3 LOOP CONTROL: `for` AND `while` STRUCTURES

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a *condition* is met, at which time the loop is terminated. There are basically two loop control constructions: `while` and `for` loops. These structures are equivalent to the **For**- and **While** constructions in the pseudocode, respectively.

4.4 THE `for` LOOP

The `for` construct is used when a block of code statements is to be executed a specified number of times. The `for`-construct has the form

```
for loop_variable = start : step : stop
    STATEMENTS
end
```

where `loop_variable` is an integer loop-control variable, the *start* and *stop* are the initial and the final values, and the *step* denotes the increment (if *step>0* or decrement (*step<0*) in the loop variable. If the *step* is omitted, then the loop variable is increased by +1. The total number of iterations done by a `for`-loop is $(stop - start + step)/step$.

Consider the following examples: Note that the range of the index variables and the values that the loop index takes in this range are given in the comment lines next to the block statements.

```
for i = 2:9
   STATEMENT(s)    % The loop-block is executed for i = 2, 3,..., 9
end
for j = 1:2:8
   STATEMENT(s)    % The loop-block is executed for i = 1, 3, 5, 7
end
for k = 5:-1:3
   STATEMENT(s)    % The loop-block is executed for k = 5, 4, 3
end
for x = 0.0:0.5:1.5
   STATEMENT(s)    % The block is executed for x = 0, 0.5, 1, 1.5
end
for a = [4, 10, 7, 12, 8]
    STATEMENTS     % The block is executed for 4, 10, 7, 12, 8
end
for j = 20:3:2
    STATEMENT(s)   % The block is skipped since 3 > 0 and 2 < 20
end
```

4.5 THE `while` LOOP

The `while` loop executes a statement or a group of statements repeatedly as long as the (controlling) *condition* is true. The `while` loop has the following form:

```
while condition
   STATEMENT(s)     % if condition is true
end
```

In this construction, *condition* is evaluated before the statement block. If the *condition* is `true`, then the block of statements will be executed. If the condition is initially `false`, the statement block will be skipped.

The following example illustrates the summation of natural numbers from 1 to 5 using a `while` structure. This code uses an accumulator (`s`) and a counter (`n`) as long as the condition (`n<6`) is satisfied. The block statements are ended with semicolon to suppress displaying the values of intermediate values of the variables. Only the result (`1+2+3+4+5=15`) is displayed.

```
s = 0              % initialize variable s
n = 1              % initialize variable n
while (n < 6)      % loop will be executed as long as n<6
   s = s + n;      % s is incremented by n
   n = n + 1;      % n is incremented by 1
end
disp(s)            % variable s is displayed, gives 15.
```

MATLAB does not have a built-in **Repeat-Until** loop like some other programming languages. However, similar functionality can be achieved using a `while` loop. The basic idea is to run a `while` loop until a specific *condition* becomes `true`.

4.6 THE `continue`, `break`, `return` STATEMENTS

The `continue` Statement: The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, continue passes control to the next iteration of the `for` or `while` loop enclosing it.

The `break` Statement: The `break` statement terminates the execution of a `for` or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside of the loop. In nested loops, `break` exits from the innermost loop only.

```
for i = 1:8
    if (i == 3 | i == 5)
        continue;      % skip iterations when i is 3 and 5
    end
    disp(i);
end
...
for i = 1:10
    If i == 5
        break;         % exit the loop when i becomes 5
    end
    disp(i);
end
```

The `return` Statement: The `return` statement is used to exit a function prematurely. That is, the `return` statement can be used to break out of (exit) the function before it has executed all of its statements in response to specific conditions and returns control to the invoking function or to the keyboard. The `return` statement is also used to terminate keyboard mode.

In the following example, the function prints `"x is negative. Exiting function."` and immediately stops the execution of the code. Any code statements after the `return` statement are skipped. The function continues to execute the rest of the statements if $x \geqslant 0$.

```
function FuncX(x)
    if ( x < 0 )
        disp('x is negative. Exiting function.');
        return;       % Exit FuncX for negative x
    end
    rest of the function statements
end
```

5 CREATING AND USING ARRAYS

Array (variable with subscripts) is a fundamental data form that MATLAB uses to store and manipulate data. A one-dimensional array, also used to represent a vector, has a row or a column of numbers. A two-dimensional array (matrix) is a list of numbers arranged in rows and columns. Arrays are useful in storing information and data in tabular form.

An array (vector) is created by typing its elements (data) inside square brackets.

```
array_name = [ elements of array ]
```

where, in the case of matrices, the semicolon is used as a separator to part the row and column elements.

5.1 ONE-DIMENSIONAL ARRAYS OR VECTORS

To create a one-dimensional array (a row vector), elements within square brackets must be separated by spaces (or commas), whereas to create a column vector, semicolons (;) are used to part the vector entries, or a row vector is transposed. For instance, the row vector $\mathbf{x} = (3, -1, 2, 0, 7)$ and the column vector $\mathbf{y} = (5, 2, 9, -6)$ are assigned as follows:

```
>> x = [ 3 -1 2 0 7 ]     % is equivalent to x = [3,-1,2,0,7]
x =
   3  -1   2   0   7
>> y = [ 5; 2; 9; -6 ]    % semicolumn denotes row separator
y =
   5
   2
   9
  -6
```

Note that the equal sign is used for assigning a variable name to a vector, and the elements of the vector (or array) separated with spaces are enclosed with square brackets. The following involves the use of transpose operation (**z'**):

```
>> z = [5, 2, 9, -6]      % commas separate row elements
z =
   5   2   9  -6
>> y = z'                 % apastrophe denotes transpose operation
y =
   5
   2
   9
  -6
```

Vectors with values uniformly spaced between the first and last terms can be constructed as follows:

```
array_name = [start:inc:stop]        % with colon operator, or
array_name =  start:inc:stop
array_name = linspace(xi:xf:nos)  % with function linspace
```

where *start*, *stop*, and *inc* denote the first value, last value, and the size of the increments, respectively. On the other hand, the `linspace()` is a MATLAB function that generates linearly spaced vectors with the number of *nos* elements between *xi* (the start value) and *xf* (the end value). Consider the following examples:

```
>> u = [3:3:16]  % generates numbers between 3 and 16 with increments of 3
u =
    3    6    9   12   15
>> a = 3:3:16    % note that a = u although [] are missing
a =
    3    6    9   12   15
>> x =[0:0.2:1]  % generates numbers between 0 and 1 with increments of 0.2
x =
    0   0.2000   0.4000   0.6000   0.8000   1.0000
>> z = linspace(0,2,6) % generates 6 numbers with equal increments in [0,2]
z =
    0   0.4000   0.8000   1.2000   1.6000   2.0000
```

Here, `u=[3:3:16]` and `x=[0:0.2:1]` generate integer and real-valued vectors, respectively. Both the colon operator or `linspace()` are effective for creating vectors with uniform spacing. The user can choose the one that best fits his needs.

5.2 TWO-DIMENSIONAL ARRAYS (MATRICES)

A two-dimensional array, also referred to as a matrix, has specified numbers in rows and columns. A matrix can be defined by using square brackets, separating elements with spaces (or commas) and rows with semicolons, as shown in the example below:

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
 16  3  2 13
  5 10 11  8
  9  6  7 12
  4 15 14  1

>> B = [6 13; 8 -2; 6 0; 14 5]
B =
    6   13
    8   -2
    6    0
   14    5
```

The use of semicolons (`;`) in matrix applications is different from the uses mentioned earlier to suppress output or to write multiple commands in a single line.

Once the matrix is entered (i.e., defined), it is automatically stored and remembered in the *Workspace*. The matrix can then be referred to as A and retrieve a particular element in a matrix by specifying its location. The location of the elements of a matrix is denoted with two indices. The element of row `i` and column `j` of the matrix A is denoted by `A(i,j)`; i.e., `A(i,j)` refers to the element $A_{i,j}$ or A_{ij}. For example, for the matrix A given above, the elements $A_{2,3}$ and $A_{4,2}$ are accessed as follows:

```
>> A(2,3)           % the 2nd row, 3rd column element
ans =
 11
>> A(4,2)           % the 4th row, 2nd column element
ans =
 15
```

Modifying a matrix or correcting a specific element with indexing is as easy as typing **`A(row,col)`**, where **`row`** and **`col`** are the row and column numbers. For the matrix A defined earlier, modifying $A_{1,3}$ and $A_{3,3}$ is carried out as follows:

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

5.3 EXTRACTING A RANGE OR SLICES FROM ARRAYS

Specific rows, columns, or sub-matrices can be extracted through slicing. A colon (`:`) operator is used to address a range of elements in arrays (*see* **Table 1.7** for its implementations). Here are some applications of the colon operator for the vector `a` and the matrix `B`:

```
>> a = [6 3 2 3 5 1 1 8 2 6 7 2 4 5 4 1]
a =
   6   3   2   3   5   1   1   8   2   6   7   2   4   5   4   1
>> a(5:9)
ans =
   5   1   1   8   2
>> B = [1 1 4; 1 3 -2; -2 1 1]
B =
   1   1   4
   1   3  -2
  -2   1   1
>> B(2,:)            % gives the second row
ans =
   1   3  -2
>> B(:,3)            % gives the third column
ans =
   4
  -2
   1
>> B(1:2,2:3)        % sub matrix from rows 1-2, columns 2-3
```

Table 1.7: Illustrating the uses of colon operator in array variables.

Operation	Description
`a(m:n)`	elements `m` through `n` of vector `a`
`B(:,n)`	elements in all the rows of column `n` of matrix `B`
`B(n,:)`	elements in all the columns of row `n` of `B`
`B(:,m:n)`	elements in all the rows between columns `m` and `n` of `B`
`B(m:n,:)`	elements in all the columns between rows `m` and `n` of `B`
`B(m:n,p:q)`	elements in rows `m` through `n` and columns `p` through `q` of `B`

```
ans =
   1   4
   3  -2
>> B(2:3,:)          % sub matrix from 2nd and 3rd rows
ans =
   1   3  -2
  -2   1   1
```

Creating a sub-matrix involves extracting a portion of a larger matrix by specifying the row and column indices of the desired section. You can use indexing with the colon operator (:) to define the range of rows and columns for the sub-matrix. The general syntax is given as

```
sub_matrix = A( r1:r2, c1:c2 );
```

where `r1` and `r2` and `c1` and `c2` are the row and column numbers of the matrix to be extracted, respectively.

Following illustrates the extraction of matrices from continuous and non-continuous rows and columns:

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1];
B = A(2:4,2:3)          % extracts rows 2 to 4, cols=2 to 3
B =
   10   11
    6    7
   15   14
C = A([1 2 4],[1 4])    % extracts rows 1, 2, and 4, cols=1 and 4
C =
   16   13
    5    8
    4    1
```

Note that the list of the indices is explicitly specified to extract specific, non-contiguous rows or columns, as in matrix C above.

5.4 INSERTING OR DELETING A ROW (OR COLUMN) INTO OR FROM MATRICES

Deleting a row or a column from a matrix is achieved by assigning an empty array `[]` to the corresponding row or column indices. After deleting a row or column, the size of the matrix is automatically updated. This procedure cannot be applied for partial deletion, i.e., one must remove either the entire row or entire column.

```
A(3,:)=[]               % removes 3rd row of matrix A given above
A =
   16    3    2   13
```

```
     5    10    11     8
     4    15    14     1
A(:,4)=[]                    % removes 4th column of updated A
A =
    16     3     2
     5    10    11
     4    15    14
```

Inserting rows or columns into a matrix can be achieved simply by using indexing and concatenation. To insert a new row into a matrix, a new row can be concatenated either before or after the existing rows. To insert a new column, a new column is similarly concatenated to the left or right of the existing columns.

```
>> A = [1, 2, 3; 7, 8, 9];  % Original matrix
>> row2 = [4, 5, 6];        % row2 is inserted as 2nd row
>> A = [A(1,:); row2; A(2:end,:)]
A =
   1   2   3
   4   5   6
   7   8   9
>> A = [1, 2, 3; 4, 5, 6];  % Original matrix
>> col2 = [7; 8];           % col2 is inserted as 2nd column
>> A = [A(:, 1), col2, A(:, 2:end)]
A =
   1   7   2   3
   4   8   5   6
```

Make sure that the new row or column to be inserted has conforming dimensions to match the existing matrix. You can also use the `insertRows` and `insertCols` functions from the File Exchange or create your own functions for more complex operations.

5.5 INSERTING ELEMENTS TO EXISTING ARRAYS

The length and the content of an existing array (one- or multi-dimensional) can be modified by inserting elements. Note that a scalar is also a vector with one element. Likewise, the number of elements of a one-dimensional array (a vector, a row or column matrix) can be increased, or it can be upscaled to be a two-dimensional matrix. Rows or columns can also be inserted into an existing matrix to obtain a matrix of different sizes. The addition of new elements is carried out by simply assigning values to the additional elements.

Consider the following examples: The vector `x` with five elements is made longer by assigning 4, 3, 2, 1 to the 6'th to 9'th elements, respectively. The vector `y` has three elements, and a new element with the value -1 is assigned to the 8'th element, an address larger than its size. MATLAB fills all the elements in between the last original element and the new element (4'th to 7'th elements) with zeros. A vector new `z`, whose 3'rd element is -5, is created with a simple assignment, i.e., `z(3)=-5`.

```
>> x = [1 2 3 4 5];  % creates a vector x
x =
   1   2   3   4   5
```

Table 1.8: Some of the special matrix creation functions in MATLAB.

Function	Description
`eye(m,n)`	Gives an $m \times n$ matrix with '1's on the main diagonal
`zeros(m,n)`	Gives an $m \times n$ matrix filled with zeros
`ones(m,n)`	Gives an $m \times n$ matrix filled with '1's
`diag(A)`	Extracts the diagonals of the matrix **A**
`rand(m,n)`	Gives an $m \times n$ matrix filled of random numbers

```
>> x(6:9)=4:-1:1     % inserts 4, 3, 2, 1 to x starting with the 6'th
x =
   1   2   3   4   5   4   3   2   1
>> y=[1 1 1]         % creates a vector y
y =
   1   1   1
>> y(8)=-1           % assign -1 to the 8'th element
y =
   1   1   1   0   0   0   0  -1
>> z(3)=-5           % assign -5 to the 3'rd element of a new array
z =
   0   0  -5
```

The following example illustrates inserting rows and columns into an existing matrix by assigning values to the new rows or columns.

```
>> a=[1 2 3 4; 5 6 7 8]
a =
   1   2   3   4
   5   6   7   8

>> a(4,:)=[1 1 1 1]
a =
   1   2   3   4
   5   6   7   8
   0   0   0   0
   1   1   1   1

>> a(:,7)=[2 2 2 2]
a =
   1   2   3   4   0   0   2
   5   6   7   8   0   0   2
   0   0   0   0   0   0   2
   1   1   1   1   0   0   2
```

Here, new elements (i.e., 1's) are assigned to the 4th row that did not exist before with `a(4,:)`. Note that the 3'rd row elements are zeroed out because they are skipped. Similarly, new elements (i.e., 2's) are assigned to the 7th column with `a(:,7)`. However, creating new rows or columns must be done carefully since the size of the rows or columns to be inserted must match the existing matrix size.

5.6 MATRIX CREATION FUNCTIONS

MATLAB provides a wide range of built-in matrix functions for performing mathematical operations and manipulations on matrices. These functions are optimized for matrix algebra and are commonly used in linear algebra, numerical methods, and data analysis. The most commonly used matrix creation functions are listed in **Table 1.8**.

Matrix creation functions help in creating special types of matrices, such as a matrix of zeros, ones, or an identity matrix. The following examples illustrate the construction of such special matrices.

The following are examples of the uses of matrix creation functions in MATLAB.

```
>> A = zeros(3,2)      % creates a 3x2 matrix full of zeros
A =
   0   0
   0   0
   0   0
>> B = ones(3,4)       % creates a 3x4 matrix full of ones
B =
   1   1   1   1
   1   1   1   1
   1   1   1   1
>> C = eye(3,3)        % creates a 3x3 identity matrix
C =
   1   0   0
   0   1   0
   0   0   1
>> D = diag([1 3 4])   % creates a 3x3 identity matrix
D =
Diagonal Matrix
   1   0   0
   0   3   0
   0   0   4
```

It should be noted that these functions with a single index (i.e., `eye(n)`, `zeros(n)`, `ones(n)`, and so on) create an $n \times n$ square matrix.

5.7 BUILT-IN FUNCTIONS FOR HANDLING ARRAYS

MATLAB provides a rich set of built-in functions for managing and handling arrays (vectors and matrices). Some of the frequently used functions are given in **Table 1.9**. These functions allow you to perform various operations, such as manipulating their shape, performing mathematical operations, and so on.

Some examples of their usage are as follows:

```
>> A = [1 3 4; 2 -1 -2; 4 1 1]  % create a 3x3 matrix
A =
   1   3   4
   2  -1  -2
   4   1   1
>> size(A)             % returns its size
ans =
   3   3
>> [m, n] = size(A)    % returns its size by setting row/col. no's
```

Table 1.9: Some of the built-in matrix functions in MATLAB.

Operation	Description
SIZE AND DIMENSIONS	
`length(A)`	gives the number of elements in A.
`size(A)`	gives a row vector [`m,n`], where m and n are the size of A.
RESHAPING AND PERMUTTING	
`reshape(A,m,n)`	creates an $m \times n$ matrix from the elements of matrix A. The elements are taken column after column. Matrix A must have $m \times n$ elements.
`transpose(A)` or `'`	transpose of matrix A
`permute(A, order)`	rearranges the dimensions of A according to the vector `order`
`sort(A)`	sorts the elements of A in ascending order
MATHEMATICAL OPERATIONS	
`dot(a,b)`	gives the dot product of two (row or column) vectors `a` and `b`
`cross(a,b)`	gives the cross product of row or column vectors `a` and `b` of size 3
`det(A)`	gives the determinant of A
`linsolve(A,b)`	gives the solution of the linear system **Ax=b**
`rank(A)`	gives the rank of matrix A
`inv(A)`	gives the inverse of matrix A
`eig(A)`	gives the eigenvalues of matrix A
`lu(A)`	gives the LU-decomposition of matrix A
LOGICAL OPERATIONS	
`A > B, A == B`	element-wise comparison of two arrays using relational operators
`any(A)`	returns true if any element of A is non-zero (or `true`)
`all(A)`	returns true if all elements of A are non-zero (or `true`)
`find(A)`	returns subscripts of elements of A are non-zero (or `true`)

```
m = 3
n = 3
>> x = [1 3 4 2 -1 -2 4 1 1]  % create an 1d array
x =
   1   3   4   2  -1  -2   4   1   1
>> length(x)          % returns the length of the array
ans = 9
>> lenX=length(x)     % returns and sets the length of the array to lenX
lenA = 3
>> C=[3 1 1 5; 4 2 -1 1; 1 2 0 3]  % a 3x4 matrix is created
C =
   3   1   1   5
   4   2  -1   1
   1   2   0   3
>> D = reshape(C,2,6)  % matrix is reshaped as 2x6
D =
   3   1   2   1   0   1
   4   1   2  -1   5   3
>> transpose(D)        % matrix D is transposed, D' also returns D^T
```

```
ans =
   3   4
   1   1
   2   2
   1  -1
   0   5
   1   3
```

MATLAB also has many built-in functions for managing mathematical vector and matrix, as well as logical operations. Some of the most common matrix operations are presented in **Table 1.9**.

```
>> A = [1 3 4; 2 -1 -2; 4 1 1];
>> det(A)        % determinant of A
  ans = -5
>> B = inv(A)    % inverse of A
A =
  -0.2000  -0.2000   0.4000
   2.0000   3.0000  -2.0000
  -1.2000  -2.2000   1.4000
>> rank(A)      % rank of A
  ans = 3
>> a=[3 1 1];b=[1 -2 4];
>> dot(a,b)    % gives the dot product of vectors a & b
  ans = 5
>> cross(a,b)  % gives the dot product of vectors a & b
  ans =
    6  -11   -7
>> c =[18;-7;8]  % create a column vector
c =
   18
   -7
    8

>> linsolve(A,c) % solve the linear system Ax=c
  ans =
   1.0000
  -1.0000
   5.0000
```

Logical operations with matrices in MATLAB allow performing element-wise comparisons and applying logical operators on matrices of the same size. Relational operators (*see* Section????) can also be used to compare corresponding elements of two matrices or a matrix and a scalar. These operations result in logical arrays, where each element is either `true (1)` or `false (0)`. The resulting logical arrays can then be used for tasks like filtering, indexing, and condition-based operations.

In the following example, the resulting matrix representing the logical values indicates that $A_{21} = B_{21}$ and $A_{31} = B_{31}$. The matrix `D` whose values are logical data indicate that A_{23} and A_{32} are less than zero. The nonzero elements of the one-dimensional array (indexed from zero) are found as 3 and 4. However, in two-dimensional arrays, the locations are counted column-wise, starting from the top. This indicates that the 4th and 7th elements are non-zero.

```
>> A = [3 2 9; 2 5 -2; 5 -1 8];
>> B = [2 7 4; 2 3  4; 5 -2 6];
>> A==B                    % Equality in case of arrays becomes
ans =
 0  0  0
 1  0  0                  % (row=2, col=1) elements are equal
 1  0  0                  % (row=3, col=1) elements are equal
>> D = A < 0               % find where A is less than zero
D =
  0  0  0
  0  0  1
  0  1  0

>> find([0 0 0 4 -1 0])  % find non-zero elements in a 1d array
ans =
   3   4
>> find([0 0 0 2; 0 3 0 0]) % find non-zero elements in a 2d array
ans =
   4
   7
```

When two scalars are compared, the result (also a scalar) is either `1` or `0`. When comparing two arrays element-by-element, the result is a logical array of the same size with `1`'s and `0`'s depending on the result of the comparison at each address.

5.8 MATHEMATICAL OPERATIONS WITH ARRAYS

Matrix arithmetic operations such as $\mathbf{A}\pm\mathbf{B}$ or $\beta\mathbf{A}$ are defined for matrices of the same size. Array (arithmetic) operations can also be carried out on an element-by-element basis. The use of period (`.`) in the operators (except in `+`) and (`-`) distinguishes the element-by-element operations from the matrix operations. A summary of array operators is given in **Table 1.4**.

Matrix operations such as multiplication, inversion, and exponentiation can be performed using the standard symbols (`*, /, ^`) or by element-by-element (`.*, ./, .^`). MATLAB additionally has left division operators (`.\` or `\`).

Let **A** and **B** are two matrices of conformable size.

```
>> A = [3 1 2; 0 -1 4; 2 2 -3]
A =
   3   1   2
   0  -1   4
   2   2  -3
>> B = [1 1 4; 1 3 -2; -2 1 1]
B =
   1   1   4
   1   3  -2
  -2   1   1
>> C=2*A - 3*B   % scalar multiplication and summation
C =
```

```
     3   -1   -8
    -3  -11   14
    10    1   -9
>> C=A*B          % matrix arithmetic multiplication
C =
     0    8   12
    -9    1    6
    10    5    1
>> D=A.*B         % element-by-element multiplication
D =
    3   1   8
    0  -3  -8
   -4   2  -3
```

The inverse of matrix $\mathbf{A}$ ($\mathbf{A}^{-1}$,) is defined such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. It can be obtained by using either the inverse function as `inv(A)` or taking `A` to the power `-1`, i.e., `A^(-1)`. These two options are illustrated in the following example:

```
>> A = [1 2 3; 0 1 4; 5 6 0]   % define a 3x3 matrix
A =
    1   2   3
    0   1   4
    5   6   0
>> B = inv(A)  % assign the inverse of A to B
  -24.0000   18.0000    5.0000
   20.0000  -15.0000   -4.0000
   -5.0000    4.0000    1.0000
>> A*B         % check A*inv(A), which should give identity matrix
ans =
    1   0   0
    0   1   0
    0   0   1
>> A^(-1)     % compute inverse A using the (-1)'th power
ans =
  -24.0000   18.0000    5.0000
   20.0000  -15.0000   -4.0000
   -5.0000    4.0000    1.0000
```

MATLAB defines multiplication of a matrix with the inverse of another matrix from the left or right as *left* or *right division* operation. *Left division* gives the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ matrix equation, i.e., $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, which may be coded in MATLAB as `x=A\b` or `x=inv(A)*b`. In the first (division) operation, `x` is found by using the *Gauss-elimination algorithm*, while in the latter the matrix inverse is based on matrix inversion followed by a matrix multiplication. *Right division* gives the solution to $\mathbf{x}\mathbf{A} = \mathbf{b}$ matrix equation, i.e., $\mathbf{x} = \mathbf{b}\mathbf{A}^{-1}$, which in Mathlab is as `x=b\A`.

```
>> A = [1 2 3; 0 1 4; 5 6 0]
>> b = [19; 21; 16]  %  define a column vector
b =
   19
```

```
   21
   16
x=A\b            % evaluate A^(-1)*b using Gauss-Elimination
>> x =
   2
   1
   5
>> c=[19 21 16]  % define a row vector
c =
   2   1   5
>> x=c/A         % evaluate c*inv(A)
x =
  -116.000    91.000    27.000
```

A matrix $\mathbf{A}$ to be inverted must be a square ($n \times n$) and non-singular ($\det(\mathbf{A}) \neq 0$) matrix. If it is singular (*non-invertible*), then MATLAB will issue a *warning* or *error message*.

6 THE SCRIPT (M-) FILES

MATLAB allows writing two kinds of program files: scripts and functions. Of these, a *script* is a program that contains a series of MATLAB commands, stored in a plain text file with an extension `.m`. That is why the scripts are often called the *M-files*. When a script file is submitted to MATLAB, the script commands are executed sequentially as if they were typed one by one from the keyboard. Scripts do not accept inputs (as arguments) and do not return any outputs. They operate on data in the MATLAB workspace.

Scripts are straightforward to create and run, making them ideal for beginners or for quickly automating tasks. Unlike function constructions, they do not require a structure with input and output arguments, thereby allowing easy assembly and execution of commands. Script files are suitable for small tasks and operate in the base workspace, meaning all variables defined within the script are directly accessible in the command window after the script has finished running. This is a useful feature, especially for quick debugging, checking intermediate results, or continuing calculations based on the script's output. Scripts are also handy when different commands or algorithms need to be quickly tested and experimented without formalizing the code into a function. In this regard, it is easy to modify and rerun parts of a script.

Scripts, however, do not encapsulate code into reusable blocks. The lack of modularity makes it harder to manage large or complex projects. If the variables inside the script file are to be run with different data, the values of the variables have to be changed manually. For more structured and reusable code, MATLAB functions are the better option.

The following 3-line script file is executed sequentially. In line 1, the array `x` of five elements is created. In line 2, the sine of every `x` is computed and stored in an array of five elements. Lines 1 and 2 terminate with a semicolon, which prevents the computed values from being displayed. In line 3, the exponential of the last array is computed element-by-element and displayed.

```
x =[0 pi/4 pi/2 3*pi/4 pi]; % array is defined, ';' suppresses output display
sin(x);                     % sine of x applies to every element of array x
exp(sin(x))   % e^sinx is computed and displayed for every element of sin(x)
```

The script output is

```
ans =
   1.0000    2.0281    2.7183    2.0281    1.0000
```

Script files can be useful for setting the global behavior of a MATLAB session (terminal and plotting parameter settings, etc.), but they should not be used for complex numerical calculations.

7 FUNCTIONS IN MATLAB

In MATLAB, functions are modules that encapsulate a code into reusable blocks that can take inputs and outputs. Functions offer modularity, reusability, and scope control, which is essential for organizing complex projects. Unlike scripts, functions have their own local workspace and do not directly interact with the base workspace unless specified.

In MATLAB, functions can be viewed in two categories: (i) *built-in functions*, those provided with the standard MATLAB software, and (ii) *user-defined functions*, those prepared by the user.

7.1 BUILT-IN FUNCTIONS

Most common functions that are available as executable programs are called built-ins. MATLAB provides a wide range of built-in functions that cover various fields like mathematics, engineering, data analysis, and visualization. These functions are pre-defined and optimized for efficient performance, allowing the users to carry out tasks without needing to manually implement algorithms from scratch. Furthermore, many of the built-in functions that come with MATLAB can be implemented in script files.

Some of the mathematical built-ins are given in **Table 1.10**. The full list of built-ins can be accessed at mathworks.com. Certain scalar MATLAB functions (trig. and inverse trig. functions, **exp**, **log**, **abs**, **sqrt**, **rem**, **round**, **floor**, **ceil**) do operate element-wise when applied to a matrix (or vector) as shown in the example below:

```
>> x =[0 pi/4 pi/2 3*pi/4 pi]   % array is defined
x =
        0   0.7854   1.5708   2.3562   3.1416
>> sin(x)           % sine of the array applies to every element
ans =
        0   0.7071   1.0000   0.7071   0.0000
>> exp(sin(x))
ans =
   1.0000   2.0281   2.7183   2.0281   1.0000
```

Other MATLAB functions operate essentially on vectors, returning a scalar value. Some of these functions are given in the table below.

```
>> arr=[3,7,0,-2,4,9,-3]      % define a 1-d array
arr =
   3   7   0  -2   4   9  -3
>> mat=[ 2 1 -1 3; 1 -4 1 2]  % define a 2-d array
mat =
   2   1  -1   3
```

Table 1.10: Some of the built-in functions.

Function	Description
`abs(x)`	absolute value of a double or complex x
`angle(x)`	polar angle
`sin(x), cos(x), tan(x)`	trigonometric functions
`asin(x), acos(x), atan(x)`	inverse trigonometric functions
`log(x), log2(x), log10(x)`	logarithm functions, e, 2, and 10 based
`sinh(x), cosh(x), tanh(x)`,...	hyperbolic functions
`asinh(x), acosh(x), atanh(x)`,...	inverse hyperbolic functions
`exp(x)`	exponential function
`sqrt(x)`	square root function
`power(x,y)`	power of a number, x^y; equivalent to `x^y`
`factorial(x)`	factorial function, integer x
`round(x)`	round to the nearest integer
`ceil(x)`	round up to next integer
`floor(x)`	round down to next integer
`sign(x)`, signum function	returns 1, -1, and 0 for $x > 0$, < 0, and $= 0$, respectively
`max(A)`	largest value of array A
`min(A)`	smallest value of array A

```
    1  -4   1   2
 >> max(arr)       % Maximum value of arr
 ans = 9
 >> max(mat)       % when applied to a 2-d array
 ans =
    2   1   1   3  % gives the row with the maximum element
 >> max(ans)       % we take advantage of the Matlab output 'ans'
 ans = 3           % gives the maximum of a 2-d array.
 >> min(arr)       % Minimum value of arr
 ans = -3
 >> mean(arr)      %  Mean value of arr
 ans = 2.5714
```

7.2 USER-DEFINED FUNCTIONS (UDFs)

User-defined functions (UDFs) in MATLAB are essential for several reasons when working on particularly complex projects, performing repeated tasks, or implementing custom algorithms. While MATLAB provides numerous built-in functions, they may not cover all specific needs, and that is where user-defined functions come in. In such cases, it is convenient to prepare and implement a user-defined function that allows the programmer to implement algorithms unique to his specific problem, giving the user complete control over how the function behaves. Once a user-defined function is prepared (i.e., written and tested), it can be used just like a built-in function.

With the use of UDFs, large or complex problems can be broken down into smaller, manageable parts. Each part can be implemented as a separate function, making the code more organized and easier to read. Functions are also all aspects of a specific task or calculation. This way, one can keep related functionality together and maintain a clean, structured code base. Having UDFs for specific tasks, the scripts as well as the main program, become more concise and readable.

These functions are subprograms that accept input arguments compatible in number and type and return the desired output arguments. Functions have their own local workspace, which isolates variables defined within the function from the base workspace. This prevents unintentional modification of variables in other parts of the program and avoids conflicts.

A user-defined function must be defined before it is used in any function or main program. The first (executable) line of a function file is called the function definition line. In this line, the function name, the number and the order of the input and output arguments are defined.

The general syntax for the user-defined function is given as follows:

```
function [out1, out2,..., outN] = functionName(in1, in2, ..., inM)
% First comment line is reported by 'lookfor'
% Additional lines of first comment are reported by 'help'
    . . .
    Matlab commands
    . . .
    out1 =  ...     % output argument-1
    . . .
    outN =  ...     % output argument-n
    . . .
end                 % end of functionName
```

where **in1, in2, ..., inM** and **out1, out2,..., outN** are the input and the returned output parameters, respectively, and **functionName** is the name of the function. All input arguments are passed by value, and any type of data, including arrays, can be implemented for either input or output arguments.

Following examples illustrate function definition lines with different combinations of input and output arguments:

```
1  function [y] = funcX(x)        % funcX is a function of x
2  function y = funcX(x)          % the same as above
3  function R = func3(x,y,z)      % a function of 3 independent variables
4  function [S,V] = cube(x,y,z)   % surface area and volume of a cube
```

Here, the function definition in line 2 is the same as the one in line 1, as one-variable functions, $y = f(x)$, can be typed without the brackets. In line 3, we have a function of 3-variables, $R = func3(x, y, z)$. In line 4, $cube(x, y, z)$ returns the surface area (**S**) and the volume (**V**) of a cube of dimensions (x, y, z).

The function body contains the computer program (code) that actually performs the computations. A well-thought-out function includes comments that explain the function, its parameters, methods used, etc. The **function** with more than one output parameter corresponds to the **Module-End Module** sub-programs of the pseudocode. If a function only has a single output argument (corresponds to the **Function-End Function** subprograms), then the square brackets may be omitted. Empty parentheses are required even if the function has no input argument. Functions normally return automatically when the **end** of the function the function.

A UDF is used just like a built-in function. The function can be called from the Command Window, from a script file, or from another function. A function can be used by setting its output to a variable (or variables), as part of a mathematical expression, or simply by typing the function name in the Command Window or a script file. In any case, the user must know exactly what the input and output arguments are. An input parameter can be a number, a variable with an assigned value, or an arithmetic expression. The parameters (arguments) are assigned according to their positions in the input and output argument lists on the function definition line.

Following examples involve the uses of the above 'functions' stated in the function definitions line:

```
>> [s, v] = cube(3, 4, 5)  % function cube is used with numbers
s = 94
v = 60
>> x = 2; y = 3; z = 5;    % pre-assigned variables
>> [s, v] = cube(x, y, z)  % function cube is used with variables
s = 62
v = 30
[a, b] = cube(x*x, 3*y, 4) % with numbers & variables
a = 176
b = 144
```

Note that the arguments of the function **cube** in line 8 involved arithmetic expressions, and the output variable names are renamed.

The following function with one independent variable is designed to evaluate $n!$ for an input integer value of **n**:

```
function f = fact(n)   % function definition
   % fact(n) returns factorial of n, i.e., n!
   f = prod(1:n);      % function body using 'prod function'
end                    % end of fact
```

Here we have used the prod function to multiply the entries of an array **[1, 2, 3, ..., n]**, which is equivalent to **f=1*2*3*...*n**. The following function with two independent variables evaluates the surface area and volume of a cylinder of radius **r** and height **h**.

```
function [area, volume] = cylinder(r,h)   % define function 'cylinder'
%for input values of r and h, the area and volume of cylinder are computed
  area = 2*pi*r*(h+r);   % calculate area
  volume = pi*r*r*h;     % calculate volume
end                      % end of function cylinder
>> [a,v] = cylinder(2.0,5.0); % set results for r=2 & h=5 to 'a' and 'v'
area = 87.965
volume = 62.832
>> fprintf('area=%10.4f volume= %10.4f',a,v)
area=   87.9646 volume=    62.8319
```

7.3 INLINE FUNCTIONS

In cases when the value of a simple algorithm or mathematical expression is required, MATLAB provides the option of using *inline (anonymous) functions*. An anonymous function is a user-defined function that is defined and used within the main body of the code.

These functions can be defined in any part of a MATLAB (in the Command Window, in script files, and inside regular user-defined functions). The general syntax for the inline functions is given as

```
function_name =  @(inp1, inp2, ..., inpn) expression
```

where **inp1**, **inp2**, ..., **inpn** are the independent variables (arguments) and **function_name** is the name of the function.

Any letter except `i` and `j` (due to denoting imaginary numbers) can be used for the independent variables. The **`expression`** may include built-in or user-defined functions. It cannot include pre-assigned variables. These functions can be used as arguments in other functions. The following are some of the inline function examples.

```
 >>  f = @(x) x^3 + x * sin(x)  % define one-variable function
f =
@(x) x ^ 3 + x * sin (x)
 >> f(pi/2)                 % evaluate f(x) for arbitrary x
 ans = 5.4466
>> g = @(x,y,z) 1/sqrt(x^2+2*y^2+3*z^2)  % define 3-variable function
 g =
@(x, y, z) 1 / sqrt (x ^ 2 + 2 * y ^ 2 + 3 * z ^ 2)
>> g(1,1,1)                % evaluate g(x,y,z) for arbitrary (x,y,z)
ans = 0.4082
```

Bibliography

[1] GILAT, A., *Matlab: An Introduction with Applications*. John Wiley, 2004.

[2] HUNT, B. R., LIPSMAN, R. L., ROSENBERG, J. *A guide to Matlab: for beginners and experienced users.* Cambridge University Press, 2005.

[3] PALM III, W. J., *Introduction to Matlab for Engineers.* McGraw Hill Education, 2011.

[4] *Matlab: The Language of Technical Computing*. Version 7. Getting started with Matlab. MathWorks, 2004.

[5] Mathworks: Official web site