



**MANIPAL INSTITUTE OF TECHNOLOGY**  
**MANIPAL**  
*(A constituent unit of MAHE, Manipal)*

# **CSE 1081: PROBLEM SOLVING USING COMPUTERS - LAB MANUAL**

**FIRST YEAR  
COMMON TO ALL BRANCHES  
(2022 CURRICULUM)**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

## CONTENTS

LAB NO.	TITLE	PAGE NO.	REMARKS
	COURSE OBJECTIVES AND OUTCOMES	i	
	EVALUATION PLAN	i	
	INSTRUCTIONS TO THE STUDENTS	ii-iii	
	SAMPLE LAB OBSERVATION NOTE PREPARATION	iii-iv	
1	SIMPLE C PROGRAMS	1	
2	BRANCHING CONTROL STRUCTURES	6	
3	LOOPING CONTROL STRUCTURES-WHILE & DO LOOPS	10	
4	LOOPING CONTROL STRUCTURES- FOR LOOPS	13	
5	1D ARRAYS	15	
6	2D ARRAYS	18	
7	STRINGS	20	
8	MODULAR PROGRAMMING - FUNCTIONS	22	
9	MATLAB INTRODUCTION & PROGRAMMING	25	
10	MODULAR PROGRAMMING - RECURSIVE FUNCTIONS, STRUCTURES, POINTERS,	43	
11	MATLAB GUI, ONLINE DOCUMENTATION WITH LIVE EDITOR	58	
12	ENGINEERING APPLICATIONS WITH MATLAB SIMULINK	55	
	REFERENCES	65	
	C LANGUAGE QUICK REFERENCE	66	

**Course Objectives:** This laboratory course enable students to

- Understand the basics of computing and various problem solving techniques
- Understand and use various programming concepts using C language
- Understand the concepts of modular programming and implement some mathematical applications
- Understand and implement Structure, Pointer and Matlab Programming

### **Course Outcomes**

On completion of this laboratory course, the students will be able to:

- 1: Understand the basics of computing and operating systems, formulate simple algorithms for arithmetic and logical problems, translate the algorithms to programs, test and execute the programs and correct syntax and logical errors.
- 2: To use arrays and pointers to formulate algorithms and simple programs. Also apply programming to solve various matrix, searching, sorting problems. Decompose a problem into functions and synthesize a complete program using divide and conquer approach. Implement recursive programming as alternate method.
- 3: Apply the concept of structure and pointer programming and MATLAB.

### **Evaluation plan**

- Internal Assessment Marks: 60%
  - ✓ Continuous Evaluation (CE) component (for each weeks experiments):10 marks
  - ✓ The CE will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
  - ✓ Total marks of the 12 weeks experiments reduced to marks out of 60
- End semester assessment of 2-hour duration: 40%

## **INSTRUCTIONS TO THE STUDENTS**

### **Pre - Lab Session Instructions**

1. Students should carry the Class notes, and lab record (King size note book with index) to every lab session
2. Be in time and follow the Instructions from Lab Instructors
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

### **In - Lab Session Instructions**

- Follow the instructions on the allotted exercises given in Lab Manual
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required
- Write the algorithm in the Lab record (required till week 4)
- Draw the corresponding flowchart in the Lab record.
  - Flowcharts need not be drawn for the programs from week 5 (1D Arrays) onwards

### **General Instructions for the exercises in Lab**

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
  - Statements within the program should be properly indented.
  - Use meaningful names for variables and functions.
  - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.

- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/she must ensure that the experiment is completed at student's end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given later in the manual as a model for observation.

## THE STUDENTS SHOULD NOT

1. Bring mobile phones or any other electronic gadgets to the lab. Students found using mobile phones and other electronic gadgets will be subjected to rules and regulation of the institution.
2. Go out of the lab without permission.

### *Sample Lab Observation Note Preparation*

#### **Title: SIMPLE C PROGRAMS**

1. Program to find area of the circle. (Hint:  $\text{Area} = 3.14 * r * r$ )

**Aim:** To write an algorithm, draw a flow chart and write a program in C to find area of a circle and verify the same with various inputs(radius).

#### **Algorithm:**

Name of the algorithm: Compute the area of a circle

Step1: Input radius

Step 2: [Compute the area]

$$\text{Area} \leftarrow 3.1416 * \text{radius} * \text{radius}$$

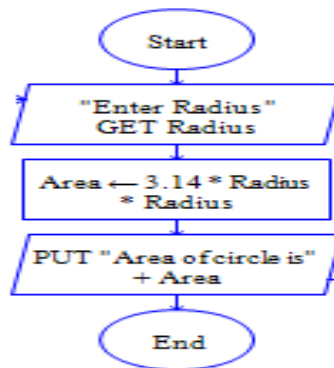
Step 3: [Print the Area]

Print 'Area of a circle =', Area

Step 4: [End of algorithm]

Stop

## Flow Chart:



## Program:

```
//student name_lab1_1.c
//program to find area of circle
#include<stdio.h>
int main()
{
    int radius;
    float area;
    printf("Enter the radius\n");
    scanf("%d", &radius);
    area=3.14*radius*radius;
    printf("The area of circle for given radius is: %f", area);
    return 0;
}
```

## Sample input and output:

The screenshot shows a Windows command prompt window titled "D:\Teaching\PSUC\Programs\area\bin\Release\area.exe". The user has entered "12" as the radius. The program outputs "The area of circle for given radius is:452.16". Below this, it shows "Process returned 0 (0x0) execution time : 9.487 s" and "Press any key to continue.".

## LAB NO.: 1

### SIMPLE C PROGRAMS

#### Objectives

In this lab, student will be able to:

1. Write C programs.
2. Compile and execute C programs.
3. Debug and trace the programs.

#### Code: Blocks Integrated Development Environment

Code: Blocks has a C editor and compiler. It allows us to create and test our programs. Code: Blocks creates Workspace to keep track of the project that is being used. A project is a collection of one or more source files. Source files are the files that contain the source code for the problem.

Let's look at C program implementation in steps by writing, storing, compiling and executing a sample program

- Create a directory with section followed by roll number (to be unique); e.g. A21.
- As per the instructions given by the lab teacher, create *InchToCm.c* program.
  - Open a new notepad file and type the given program
- Save the file with name and extension as "*InchToCm.c*" into the respective directory created.

#### Sample Program (*InchToCm.c*):

```
// InchToCm.c
#include <stdio.h>
int main()
{
    float centimeters, inches;
    printf( "This program converts inches to centimeters"\n);
    printf( "Enter a number");
    scanf(“%f”, &inches);
    centimeters = inches * 2.54;
    printf(“%f inches is equivalent to %f centimeters\n”, inches, centimeters);
    return 0;
} // end main
```

3. Run the program as per the instructions given by the lab teacher.
  - Compile the saved program and run it either by using keyboard short cuts or through the menu.

## PROGRAM STRUCTURE AND PARTS

### *Comments*

The first line of the file is:

```
// InchToCm.c
```

This line is a comment. Let's add a comment above the name of the program that contains the student's name.

- Edit the file. Add the student's name on the top line so that the first two lines of the file now look like:

```
// student's name
```

```
// InchToCm.c
```

Comments tell people reading the program something about the program. The compiler ignores these lines.

### *Preprocessor Directives*

After the initial comments, the student should be able to see the following lines:

```
#include <stdio.h>
```

This is called a preprocessor directive. It tells the compiler to do something. Preprocessor directives always start with a # sign. The preprocessor directive includes the information in the file *stdio.h*. as part of the program. Most of the programs will almost always have at least one include file. These header files are stored in a library that shall be learnt more in the subsequent labs.

### *The function main ()*

The next non-blank line *int main ()* gives the name of a function. There must be exactly one function named *main* in each C program, and *main* is the function where program execution starts when the program begins running. The *int* before *main ()* indicates the function is returning an integer value and also to indicate empty argument list to the function. Essentially functions are units of C code that do a particular task. Large programs will have many functions just as large organizations have many functions. Small programs, like smaller organizations, have fewer functions. The parentheses following the words *int main* contains a list of arguments to the function. In the case of



this function, there are no *arguments*. Arguments to functions tell the function what objects to use in performing its task. The curly braces ({} on the next line and on the last line (}) of the program determine the beginning and ending of the function.

### ***Variable Declarations***

The line after the opening curly brace, *float centimeters, inches;* is called a variable declaration. This line tells the compiler to reserve two places in memory with adequate size for a real number (the *float* keyword indicates the variable as a real number). The memory locations will have the names *inches* and *centimeters* associated with them. The programs often have many different variables of many different types.

## **EXECUTABLE STATEMENTS**

### ***Output and Input***

The statements following the variable declaration up to the closing curly brace are executable statements. The executable statements are statements that will be executed when the program run. `printf()` statement tells the compiler to generate instructions that will display information on the screen when the program run, and *scanf()* statement reads information from the keyboard when the program run.

<b>Format Specifiers</b>	<b>Description</b>	<b>Example</b>
<code>%d</code>	<code>%d</code> is used to print the value of integer variable. We can also use <code>%i</code> to print integer value. <code>%d</code> and <code>%i</code> have same meaning	<pre>int v; scanf("%d",&amp;v); printf("value is %d", v);</pre>
<code>%f</code>	<code>%f</code> is used to print the value of floating point variable in a decimal form.	<pre>float v; scanf("%f",&amp;v); printf("value is %f",v);</pre>
<code>%c</code>	<code>%c</code> is used to print the value of character variable.	<pre>char ch; scanf("%c",&amp;v); printf("value is %c",ch)</pre>
<code>%s</code>	<code>%s</code> is used to print the string	<pre>char name[20]; scanf("%s",v); printf("value is %s",name)</pre>

### ***Assignment Statements***

The statement *centimeters = inches \* 2.54;* is an assignment statement. It calculates what is on the right hand side of the equation (in this case *inches \* 2.54*) and stores it in the memory location that has the name specified on the left hand side of the equation (in this case, *centimeters*). So *centimeters = inches \* 2.54* takes whatever was read into the memory location *inches*, multiplies it by 2.54, and stores the result in *centimeters*. The next statement outputs the result of the calculation.

### ***Return Statement***

The last statement of this program, *return 0;* returns the program control back to the operating system. The value 0 indicates that the program ended normally. The last line of every main function written should be *return 0;*

### ***Syntax***

Syntax is the way that a language must be phrased in order for it to be understandable. The general form of a C program is given below:

```
// program name
// other comments like what program does and student's name
#include <appropriate files>
int main()
{
    Variable declarations;
    Executable statements;
} // end main
```

### **Lab Exercises**

1. Write a C program to add two integers a and b read through the keyboard. Display the result using third variable sum
2. Write a C program to find the sum, difference, product and quotient of 2 numbers.
3. Write a C program to print the ASCII value of a character
4. Write a C program to display the size of the data type int, char, float, double, long int and long double using size of ( ) operator.
5. Input P, N and R to compute simple and compound interest. [Hint:  $SI = PNR/100$ ,  $CI = P(1+R/100)^N - P$ ]

6. Input radius to find the volume and surface area of a sphere. [Hint: volume =  $(4\pi r^3)/3$ , Area= $4\pi r^2$ ]
7. Convert the given temperature in Fahrenheit to Centigrade. [Hint:  $C=5/9(F-32)$ ]
8. Write a C program to evaluate the following expression for the values  $a = 30$ ,  $b=10$ ,  $c=5$ ,  $d=15$ 
  - (i)  $(a + b) * c / d$     (ii)  $((a + b) * c) / d$
  - (iii)  $a + (b * c) / d$     (iv)  $(a + b) * (c / d)$

### **Additional Exercises**

1. Write a C program to convert given number of days into years , weeks and days.
2. Convert the time in seconds to hours, minutes and seconds. [Hint: 1 hr =3600 sec]
3. Write a C program for the following
  - a)  $ut + 1/2 at^2$     b) (ii)  $a^2 + 2ab + b^2$
4. Determine how much money (in rupees) is in a piggy bank that contains denominations of 20, 10 and 5 rupees along with 50 paisa coins. Use the following values to test the program: 13 twenty rupee notes, 11 ten rupee notes, 7 five rupee coins and 13 fifty paisa coins.  
 [Hint:  $13 * 20 + 11 * 10 + 7 * 5 + 0.50 * 13 = \text{Rs.}411.50$ ].

## LAB NO.: 2

# BRANCHING CONTROL STRUCTURES

### Objectives:

In this lab, student will be able to do C programs using

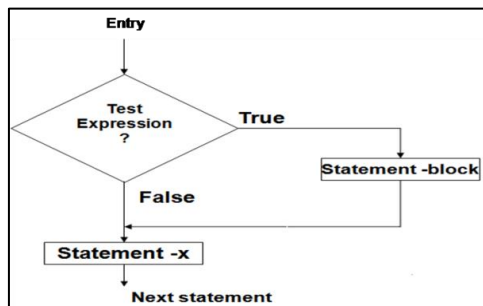
1. **simple *if*** statement
2. ***if-else*** statement
3. ***switch-case*** statement

### Introduction:

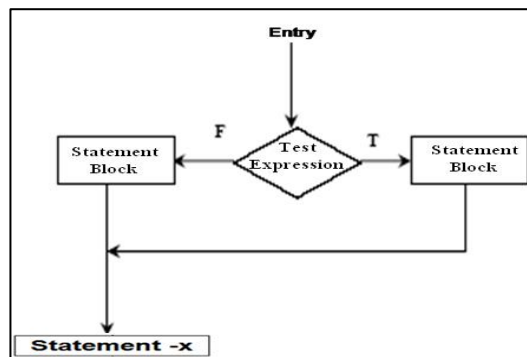
- A control structure refers to the way in which the programmer specifies the order of execution of the instructions

C decision making and branching statements flow control actions:

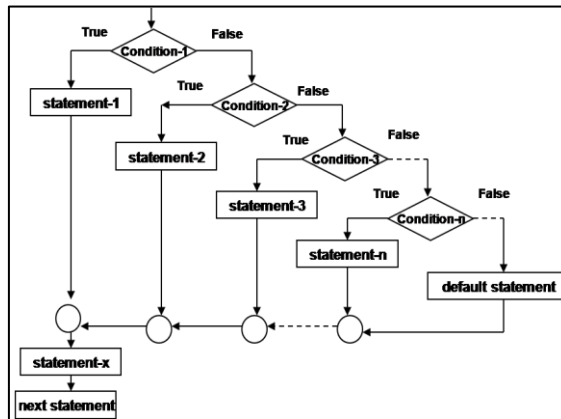
### Simple if statement:



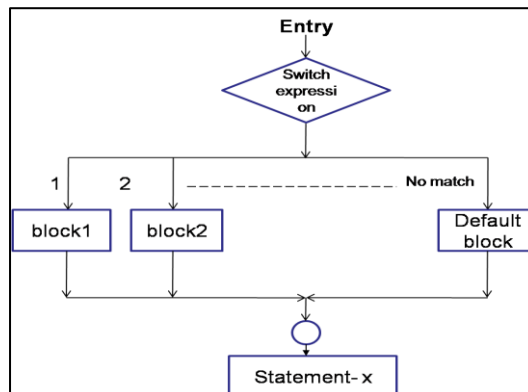
### If - else statement:



## Else - if ladder:



## Switch statement:



## Solved Exercise

C program to compute all the roots of a quadratic equation

```
#include<stdio.h>
```

```
#include<math.h>
```

```
int main() {
```

```
int a,b,c;
```

```
float root1, root2, re, im, disc;
```

```
scanf("%d,%d,%d",&a,&b,&c);
```

```
disc=b*b-4*a*c;
```

```
if (disc<0) // first if condition
```

```
{
```

```

printf("imaginary roots\n");
re= - b / (2*a);
im = pow(fabs(disc),0.5)/(2*a);
printf("%f +i %f",re,im);
printf("%f -i %f",re,im);
//printf("%f, re", "+ i",im);
//cout<<re<<"-i"<<im;
}
else if (disc==0){ //2nd else-if condition
printf("real & equal roots");
re=-b / (2*a);
printf("Roots are %f",re);
}
else{ /*disc > 0- otherwise part with else*/
printf("real & distinct roots");
printf("Roots are");
root1=(-b + sqrt(disc))/(2*a);
root2=(-b - sqrt(disc))/(2*a);
printf("%f and %f",root1,root2);
}
return 0;
}

```

## Lab Exercises

With the help of various branching control constructs like *if*, *if-else* and *switch* case statements,

Write C programs to do the following:

1. Check whether the given number is odd or even
2. Find the largest among given 3 numbers
3. Compute all the roots of a quadratic equation using *switch case* statement.  
[Hint:  $x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$ ]
4. Find the smallest among three numbers using conditional operator.

### Additional Exercises

1. Check whether the given number is zero, positive or negative, using *else-if* ladder.
2. Accept the number of days a member is late to return the book. Calculate and display the fine with the appropriate message using if-else ladder. The fine is charged as per the table below:

Late period	Fine
5 days	Rs. 0.50
6 – 10 days	Rs. 1.00
Above 10 days	Rs. 5.00
After 30 days	Rs. 10.00

3. Write a program that will read the value of x and evaluate the following function

$$Y = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

Use else if statements & Print the result ('Y' value).

## LAB NO.: 3

# LOOPING CONTROL STRUCTURES-WHILE & DO LOOPS

### Objectives:

In this lab, student will be able to:

1. Write and execute C programs using 'while' statement
2. Write and execute C programs using 'do-while' statement
3. To learn to use break and continue statements in while and do while loop statements.

### Introduction:

- Iterative (repetitive) control structures are used to repeat certain statements for a specified number of times.
- The statements are executed as long as the condition is true
- These types of control structures are also called as loop control structures
- Three kinds of loop control structures are:
  - while
  - do-while
  - for

### C looping control structures:

#### While loop:

```
while (test condition)
{
    body of the loop
}
```

#### Do-while loop:

```
do
{
    body of the loop
}
while (test condition);
```

### Solved Exercise

[Understand the working of looping with this illustrative example for finding sum of natural numbers up to 100 using **while** and **do-while** statements]

#### Using **do-while**

```
#include<stdio.h>
int main()
{
    int n;
    int sum;
```



```

sum=0; //initialize sum
n=1;
do
{
sum = sum + counter;
counter = counter +1;
} while (counter < 100);
printf(“%d”,sum);
return 0;}

```

### Using **while**

```

#include<stdio.h>
int main( )
{
int n;
int sum;
sum=0; //initialize sum
n=1;
while (n<100)
{
sum = sum + n;
n = n +1;
}
printf(“%d”,sum);
return 0; }

```

### **Lab Exercises**

Write C programs to do the following with the help of iterative (looping) control structures such as **while** and **do-while** statements

1. Reverse a given number and check if it is a palindrome or not. (use while loop).  
[Ex: 1234, reverse= $4 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0 = 4321$ ]
2. Generate prime numbers between 2 given limits.(use while loop)
3. Check if the sum of the cubes of all digits of an inputted number equals the number itself (Armstrong Number). (use while loop)

4. Write a program using do-while loop to read the numbers until -1 is encountered. Also count the number of prime numbers and composite numbers entered by user. [Hint: 1 is neither prime nor composite]

**Additional Exercises:**

1. Check whether the given number is strong or not.  
[Hint: Positive number whose sum of the factorial of its digits is equal to the number itself] Ex:  $145 = 1! + 4! + 5! = 1 + 24 + 120 = 145$  is a strong number.
2. Write a program to demonstrate use of break and continue statements in while and do-while loops.

## LAB NO.: 4

### LOOPING CONTROL STRUCTURES- FOR LOOPS

#### Objectives:

In this lab, student will be able to:

- Write and execute C programs using ‘for’ statement
- To learn to use break and continue statements in for loop statements.

#### Introduction:

- For loop statements are used to repeat certain statements for a specified number of times.
- The statements are executed as long as the execution condition is true
- These types of control structures are also called as loop control structures

#### For loop:

```
for (initialization; test condition; increment/decrement)
{
    body of the loop
}
```

#### Lab Exercises

##### With the help of *for loop* statements,

1. Generate the multiplication table for ‘*n*’ numbers up to ‘*k*’ terms (using nested for loops).

```
[ Hint: 1  2  3  4  5  ....  k
        2  4  6  8 10 ...2*k
        .....
        n..... n*k ]
```

2. Generate Floyd’s triangle using natural numbers for a given limit N. (using for loops)

[Hint: Floyd’s triangle is a right angled-triangle using the natural numbers]

Ex: Input: N = 4

Output:

```
1
2 3
4 5 6
7 8 9 10
```

3. Evaluate the sine series,  $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$  to n terms.
4. Check whether a given number is perfect or not.  
[Hint: Sum of all positive divisors of a given number excluding the given number is equal to the number] Ex:  $28 = 1 + 2 + 4 + 7 + 14 = 28$  is a perfect number

**Additional Exercises:**

1. Find out the generic root of any number.  
[Hint: Generic root is the sum of digits of a number until a single digit is obtained.]  
Ex: Generic root of 456 is  $4 + 5 + 6 = 15 = 1 + 5 = 6$
2. Write a program to demonstrate use of break and continue statements in *for* loop

## LAB NO.: 5

# 1D ARRAYS

### Objectives:

In this lab, student will be able to:

- Write and execute programs on 1Dimensional arrays

### Introduction to 1D Arrays

#### 1 Dimensional Array

#### Definition:

- An array is a group of related data items that share a common name.
- The array elements are placed in a contiguous memory location.
- A particular value in an array is indicated by writing an integer number called index number or subscript in square brackets after the array name. The least value that an index can take in array is 0.

#### Array Declaration:

***data-type name [size];***

- ✓ where data-type is a valid data type (like int, float,char...)
- ✓ name is a valid identifier
- ✓ size specifies how many elements the array has to contain
- ✓ size field is always enclosed in square brackets [ ] and takes static values.

#### Total size of 1D array:

The Total memory that can be allocated to 1D array is computed as:

Total size =size \*(sizeof(data\_type));

where, *size* is number of elements in 1-D array

*data\_type* is basic data type.

*Sizeof()* is an unary operator which returns the size of expression or data type in bytes.

For example, to represent a set of 5 numbers by an array variable ***Arr***, the declaration the variable ***Arr*** is

***int Arr[5];***

## Solved Exercise

Sample Program to read  $n$  elements into a 1D array and print it:

```
#include<stdio.h>
int main()
{
    int a[10], i, n;
    printf("enter no of elements");
    scanf("%d",&n);
    printf("enter n values\n");
    for(i=0;i<n;i++) // input 1D array
        scanf("%d",&a[i]);
    printf("\nNumbers entered are:\n");
    for(i=0;i<n;i++) // output 1D array
        printf("%d",a[i]);
    return 0;
}
```

## Output:

```
Enter no. of elements
3
Enter n values
9
11
13

Numbers entered are:
9
11
13
```

## Lab Exercises

With the knowledge of 1D arrays, Write C programs to do the following:

1. Find the largest and smallest element in a 1D array.
2. Arrange the given elements in a 1D array in ascending and descending order using bubble sort method. [Hint: use switch case (as case 'a' and case 'd') to specify the order].

3. Insert an element into a 1D array, by getting an element and the position from the user.
4. Search the position of the number that is entered by the user and delete that particular number from the array and display the resultant array elements.

### **Additional Exercises on 1D array**

1. Print all the prime numbers in a given 1D array.
2. Search an element in a 1D array using linear search.
3. Delete all the occurrences of the element present in the array which is inputted by the user.
4. Write a program to enter number of digits and create a number using this digit.

[Hint: Input: Enter number of digits: 3

Enter units' place digit: 1, Enter tens place digit: 2, Enter hundreds place digit: 5

The number is 521]

## LAB NO.: 6

### 2D ARRAYS

#### Objectives:

In this lab, student will be able to:

- Write and execute programs on 2D dimensional arrays

#### Introduction to 2 Dimensional Arrays

- It is an ordered table of homogeneous elements.
- It can be imagined as a two dimensional table made of elements, all of them of a same uniform data type.
- It is generally referred to as matrix, of some rows and some columns. It is also called as a two-subscripted variable.

For example

```
int marks[5][3];
```

```
float matrix[3][3];
```

```
char page[25][80];
```

- The first example tells that marks is a 2-D array of 5 rows and 3 columns.
- The second example tells that matrix is a 2-D array of 3 rows and 3 columns.
- Similarly, the third example tells that page is a 2-D array of 25 rows and 80 columns.

#### Solved Exercise:

```
#include<stdio.h>
int main()
{
    int i,j,m,n,a[100][100];
    printf("enter dimension of matrix");
    scanf("%d %d",&m,&n);
    printf("enter the elements");
    for(i=0;i<m;i++) // input 2D array using 2 for loops
    {
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    }
}
```



```

    }
    for(i=0;i<m;i++) // output 2D array with 2 for loops
    {
        for(j=0;j<n;j++)
            printf("%d\t",a[i][j]);
        printf("\n");
    }
    return 0;
}

```

### Lab Exercises

With the knowledge of 2D arrays,

Write C programs to do the following:

1. Find whether a given matrix is symmetric or not. [Hint:  $A = A^T$ ]
2. Find the trace and norm of a given square matrix.  
[Hint: Trace= sum of principal diagonal elements  
Norm= SQRT (sum of squares of the individual elements of an array)]
3. Perform matrix multiplication.
4. To interchange the primary and secondary diagonal elements in the given Matrix.

### Additional Exercises on 2D arrays

1. Interchange any two Rows & Columns in the given Matrix.
2. Search for an element in a given matrix and count the number of its occurrences.
3. Compute the row sum and column sum of a given matrix.
4. Check whether the given matrix is magic square or not.
5. Check whether the given matrix is a Lower triangular matrix or not.

Ex:     1  0  0  
          2  3  0  
          4  5  6

## LAB NO.: 7

# STRINGS

### Objectives:

In this lab, student will be able to:

1. Declare, initialize, read and write a string
2. Write C programs with and without string handling functions to manipulate the given string

### Introduction

- A string is an array of characters.
- Any group of characters (except double quote sign) defined between double quotations is a constant string.
- Character strings are often used to build meaningful and readable programs.

The common operations performed on strings are

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings to another
- Extracting a portion of a string etc.

### Declaration

Syntax: ***char string\_name[size];***

- The size determines the number of characters in the string\_name.

### Solved Exercise

Program to read and display a string

```
#include<stdio.h>
int main()
{
    const int MAX = 80; //max characters in string
    char str[MAX];      //string variable str
    printf("Enter a string: ");
    scanf("%s",str);
```

```
//put string in str
Printf(“You entered: %s\n” str); //display string from str
return 0;}
```

## Lab Exercises

With the brief introduction and knowledge on strings, write C programs without using STRING-HANDLING functions for the following:

1. Count the number of words in a sentence.
2. Input a string and toggle the case of every character in the input string.

Ex:           INPUT: aBcDe

              OUTPUT: AbCdE

3. Check whether the given string is a palindrome or not.
4. Arrange ‘n’ names in alphabetical order (hint: use string handling function-*strcpy*)
5. Delete a word from the given sentence.

Ex: INPUT: I AM STUDYING IN MIT

      TO BE DELETED: STUDYING

      OUTPUT: I AM IN MIT

## Additional Exercises

1. Search for a given substring in the main string.
2. Delete all repeated words in the given String.
3. Read a string representing a password character by character and mask every character in the input with ‘\*’.
4. Write a C program using 1D array to read an alphanumeric string (Eg. abc14fg67) and count the number of characters and digits in the given string and display the sum of all the digits.

## LAB NO.: 8

# MODULAR PROGRAMMING -FUNCTIONS

### Objectives:

In this lab, student will be able to:

1. Understand modularization and its importance
2. Define and invoke a function
3. Analyze the flow of control in a program involving function call
4. Write programs using functions

### Introduction

- A **function** is a set of instructions to carry out a particular task.
- Using functions programs can be structured in a **more modular** way.

### Function definition and call

```
// FUNCTION DEFINITION
Return type  Function name  Parameter List
void DisplayMessage(void)
{
    cout << "Hello from function DisplayMessage\n";
}

int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

## Solved Exercise

1. Program for explaining concept of multiple functions

```
#include<stdio.h>
void First (void){
    printf("I am now inside function First\n");
}
void Second (void){
    printf("I am now inside function Second\n");
    First();
    printf("Back to Second\n");
}
int main (){
    printf("I am starting in function main\n");
    First ();
    printf("Back to main function \n");
    Second ();
    printf("Back to main function \n");
    return 0;
}
```

## Lab Exercises

With the knowledge of modularization, function definition, function call etc., write C programs which implement simple functions.

Write C programs as specified below:

1. Write a function **Fact** to find the factorial of a given number. Using this function, compute **NCR** in the main function.
2. Write a function **Largest** to find the maximum of a given list of numbers. Also write a main program to read N numbers and find the largest among them using this function.
3. Write a function **IsPalin** to check whether the given string is a palindrome or not. Write a main function to test this function.
4. Write a function **CornerSum** which takes as a parameter, no. of rows and no. of columns of a matrix and returns the sum of the elements in the four corners of the matrix. Write a main function to test the function.

## Additional Exercises

1. Write a function **IsPrime** to check whether the given number is prime or not. Using this function, generate first N prime numbers in the main function.
2. Write a function **array\_sum** to find the sum of 'n' numbers in an array. Write a main program to read 'n' numbers and use **array\_sum** function to find the sum of 'n' numbers.
3. Write a function **toggle** to toggle the case of each character in a sentence. Write a main program to read a sentence and use **toggle** function to change the case of each character in the given sentence.

## **LAB NO.: 9**

### **MATLAB INTRODUCTION & PROGRAMMING**

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or Fortran.

The name MATLAB stands for matrix laboratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB engines incorporate the LAPACK and BLAS libraries, embedding the state of the art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of add-on application-specific solutions called toolboxes. Very important to most users of MATLAB, toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems.

Some of the areas in which **toolboxes** are available include

- **Signal processing:** This Toolbox provides functions and applications to generate, measure, transform, filter, and visualize signals.
- **Control systems:** Provides industry-standard algorithms and applications for systematically analysing, designing, and tuning linear control systems.
- **Neural networks:** Provides functions and apps for modeling complex nonlinear systems that are not easily modelled with a closed-form equation. Neural Network Toolbox supports supervised learning with feed-forward, radial basis, and dynamic networks. It also supports unsupervised learning with self-organizing maps and competitive layers
- **Fuzzy logic:** This toolbox lets you model complex system behaviours using simple logic rules, and then implement these rules in a fuzzy inference system.
- **Wavelets:** Provides functions and an application for developing wavelet-based algorithms for the analysis, synthesis, denoising, and compression of signals and images. The toolbox lets you explore wavelet properties and applications such as speech and audio processing, image and video processing, biomedical imaging, and 1-D and 2-D applications in communications and geophysics.

### ***The MATLAB System***

The MATLAB system consists of five main parts:

1. **Development Environment:** This is the set of tools and facilities that help you use MATLAB functions and files. Many of these tools are graphical user interfaces. It includes the MATLAB desktop and Command Window, a command history, an editor and debugger, and browsers for viewing help, the workspace, files, and the search path.
2. **The MATLAB Mathematical Function Library:** This is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.
3. **The MATLAB Language:** This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both “programming in the small” to rapidly create quick and dirty throw-away programs, and “programming in the large” to create large and complex application programs.



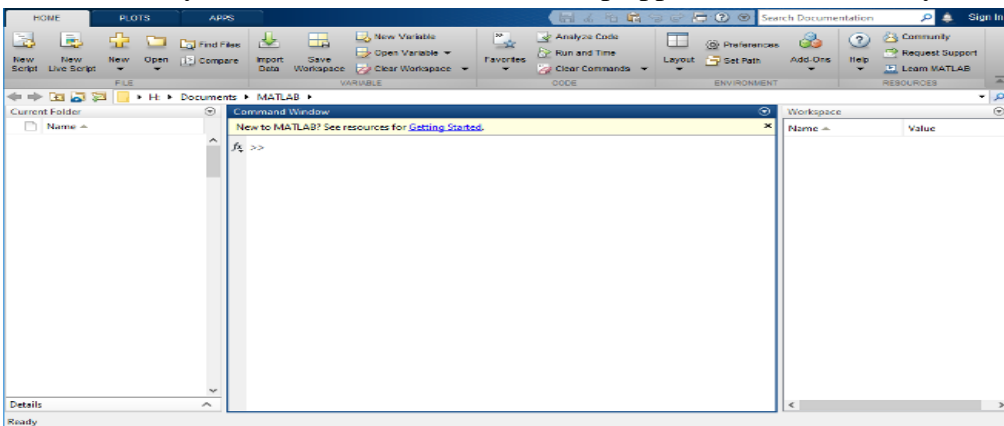
4. **Graphics:** MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications.
5. **The MATLAB External Interfaces/API:** This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

## Features

- Basic data element: *matrix*
  - Auto dimensioning
  - eliminates data type declarations
  - ease of programming
  - Operator overloading
  - Case sensitive
- Advanced visualisation
  - 2D and 3D
  - Simple programming
  - colour graphics
- Open environment

## Desktop basics

When you start MATLAB®, the desktop appears in its default layout.



The desktop includes these panels:

**Current Folder** — Access your files.

**Command Window** — Enter commands at the command line, indicated by the prompt (`>>`).

**Workspace** — Explore data that you create or import from files.

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named `a` by typing this statement at the command line:

```
a = 1
```

MATLAB adds variable `a` to the workspace and displays the result in the Command Window.

```
a =
```

```
1
```

Create a few more variables.

```
b = 2
```

```
b =
```

```
2
```

```
c = a + b
```

```
c =
```

```
3
```

```
d = cos(a)
```

```
d =
```

```
0.5403
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of your calculation.

```
sin(a)
```

```
ans =
```

```
0.8415
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

```
e = a*b;
```

You can recall previous commands by pressing the up- and down-arrow keys, ↑ and ↓. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command `b = 2`, type `b`, and then press the up-arrow key.

## Matrices and Arrays

*MATLAB* is an abbreviation for "matrix laboratory." While other programming languages mostly work with numbers one at a time, MATLAB® is designed to operate primarily on whole matrices and arrays.

All MATLAB variables are multidimensional *arrays*, no matter what type of data.

A *matrix* is a two-dimensional array often used for linear algebra.

### Array Creation

To create an array with four elements in a single row, separate the elements with either a comma (,) or a space.

```
a = [1 2 3 4]
```

```
a = 1×4
```

```
1    2    3    4
```

This type of array is a *row vector*.

To create a matrix that has multiple rows, separate the rows with semicolons.

```
a = [1 2 3; 4 5 6; 7 8 10]
```

```
a = 3×3
```

```
1    2    3
4    5    6
7    8   10
```

Another way to create a matrix is to use a function, such as `ones`, `zeros`, or `rand`. For example, create a 5-by-1 column vector of zeros.

```
z = zeros(5,1)
```

$$z = 5 \times I$$

0  
0  
0  
0  
0

## Matrix and Array Operations

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

$$a + 10$$

$$\text{ans} = 3 \times 3$$

11 12 13  
14 15 16  
17 18 20

$$\sin(a)$$

$$\text{ans} = 3 \times 3$$

0.8415 0.9093 0.1411  
-0.7568 -0.9589 -0.2794  
0.6570 0.9894 -0.5440

To transpose a matrix, use a single quote ('):

$$a'$$

$$\text{ans} = 3 \times 3$$

1 4 7  
2 5 8  
3 6 10

You can perform standard matrix multiplication, which computes the inner products between rows and columns, using the `*` operator. For example, confirm that a matrix times its inverse returns the identity matrix:

```
p = a*inv(a)
```

```
p = 3×3
```

```
1.0000    0    -0.0000
0    1.0000    0
0    0    1.0000
```

Notice that `p` is not a matrix of integer values. MATLAB stores numbers as floating-point values, and arithmetic operations are sensitive to small differences between the actual value and its floating-point representation. You can display more decimal digits using the `format` command:

```
format long
```

```
p = a*inv(a)
```

```
p = 3×3
```

```
1.0000000000000000    0    -0.0000000000000000
0    1.0000000000000000    0
0    0    0.9999999999999998
```

Reset the display to the shorter format using

```
format short
```

`format` affects only the display of numbers, not the way MATLAB computes or saves them.

To perform element-wise multiplication rather than matrix multiplication, use the `.*` operator:

```
p = a.*a
```

```
p = 3×3
```

```
1    4    9
16   25   36
49   64  100
```

The matrix operators for multiplication, division, and power each have a corresponding array operator that operates element-wise. For example, raise each element of `a` to the third power:

```
a.^3
```

```
ans = 3×3
```

```

1      8      27
64     125    216
343    512    1000
```

## Concatenation

*Concatenation* is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets `[]` is the concatenation operator.

```
A = [a,a]
```

```
A = 3×6
```

```

1  2  3  1  2  3
4  5  6  4  5  6
7  8 10  7  8 10
```

Concatenating arrays next to one another using commas is called *horizontal* concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate *vertically* using semicolons.

```
A = [a; a]
```

```
A = 6×3
```

```

1  2  3
4  5  6
7  8 10
1  2  3
4  5  6
7  8 10
```

## Complex Numbers

Complex numbers have both real and imaginary parts, where the imaginary unit is the square root of -1.

`sqrt(-1)`

`ans = 0.0000 + 1.0000i`

To represent the imaginary part of complex numbers, use either `i` or `j`.

`c = [3+4i, 4+3j; -i, 10j]`

`c = 2×2 complex`

`3.0000 + 4.0000i   4.0000 + 3.0000i`

`0.0000 - 1.0000i   0.0000 + 10.0000i`

## Operators

- **Arithmetic operators**

`+` : Addition      `-` : Subtraction

`*` : Multiplication   `/` : Division

`\` : Left Division   `^` : Power

- **Relational operators**

`<` : Less than      `<=` : Less than or equal to

`>` : Greater than      `>=` : Greater than or equal to

`==` : Equal      `~=` : Not equal

- **Logical operators**

`&` : AND      `|` : OR      `~` : NOT

- **Array operations**

Matrix operations preceded by a `.`(*dot*) indicates array operation.

- **Simple math functions**

`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh` ....

`log`, `log10`, `exp`, `sqrt` ...

- **Special constants**

`pi`, `inf`, `i`, `j`, `eps` .....

## Control Flow Statements

- **for loop** :       for k = 1:m  
                      .....  
                      end
- **while loop** :       while *condition*  
                      .....  
                      end
- **if statement** :     if *condition1*  
                          .....  
                      else if *condition2*  
                          .....  
                      else  
                          .....  
                      end

## Array indexing

Every variable in MATLAB® is an array that can hold many numbers. When you want to access selected elements of an array, use indexing.

For example, consider the 4-by-4 magic square A:

```
A = magic(4)
```

```
A = 4×4
```

```
16   2   3  13
 5  11  10   8
 9   7   6  12
 4  14  15   1
```

There are two ways to refer to a particular element in an array. The most common way is to specify row and column subscripts, such as

```
A(4,2)
```

```
ans = 14
```



Less common, but sometimes useful, is to use a single subscript that traverses down each column in order:

```
A(8)
```

```
ans = 14
```

Using a single subscript to refer to a particular element in an array is called *linear indexing*.

If you try to refer to elements outside an array on the right side of an assignment statement, MATLAB throws an error.

```
test = A(4,5)
```

Index exceeds matrix dimensions.

However, on the left side of an assignment statement, you can specify elements outside the current dimensions. The size of the array increases to accommodate the newcomers.

```
A(4,5) = 17
```

```
A = 4×5
```

```
16   2   3  13   0
 5  11  10   8   0
 9   7   6  12   0
 4  14  15   1  17
```

To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form start:end. For example, list the elements in the first three rows and the second column of A:

```
A(1:3,2)
```

```
ans = 3×1
```

```
2
11
7
```

The colon alone, without start or end values, specifies all of the elements in that dimension. For example, select all the columns in the third row of A:

```
A(3,:)
```

```
ans = 1×5
```

```
9   7   6  12   0
```

The colon operator also allows you to create an equally spaced vector of values using the more general form start:step:end.

```
B = 0:10:100
```

```
B = 1:11
```

```
0 10 20 30 40 50 60 70 80 90 100
```

If you omit the middle step, as in start:end, MATLAB uses the default step value of 1.

## Workspace variables

The *workspace* contains variables that you create within or import into MATLAB® from data files or other programs. For example, these statements create variables A and B in the workspace.

```
A = magic(4);
```

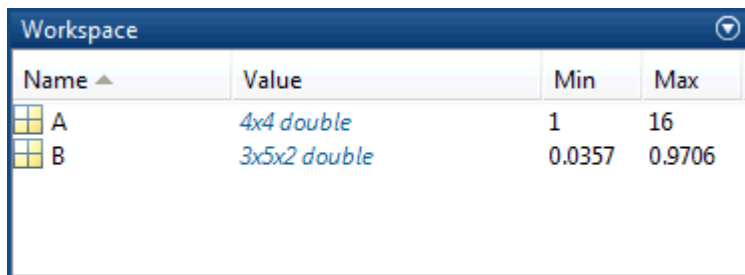
```
B = rand(3,5,2);
```

You can view the contents of the workspace using whos.

```
whos
```

Name	Size	Bytes	Class	Attributes
A	4x4	128	double	
B	3x5x2	240	double	

The variables also appear in the Workspace pane on the desktop.



Workspace variables do not persist after you exit MATLAB. Save your data for later use with the save command,

```
save myfile.mat
```

Saving preserves the workspace in your current working folder in a compressed file with a .mat extension, called a MAT-file.

To clear all the variables from the workspace, use the clear command.

Restore data from a MAT-file into the workspace using load.

load myfile.mat

### Some useful commands

who whos clc clf clear load save pause help pwd ls dir

## Getting started with MATLAB

- Using **Windows Explorer**, create a folder *user\_name* in the directory *c:\cslab\batch\_index*

For uniformity, let the *batch\_index* be *A1*, *A2*, *A3*, *B1*, *B2*, or *B3* and *user\_name* be the *roll number*

- Invoke **MATLAB**

Running MATLAB creates one or more windows on your monitor. Of these the **Command Window** is the primary place where you interact with MATLAB. The prompt `>>` is displayed in the Command window and when the Command window is active, a blinking cursor should appear to the right of the prompt.

## Interactive Computation, Script files

**Objective:** Familiarise with MATLAB Command window, do some simple calculations using array and vectors.

### Exercises

The basic variable type in MATLAB is a matrix. To declare a variable, simply assign it a value at the MATLAB prompt. Let's try the following examples:

#### 1. Elementary matrix/array operations

To enter a row vector

```
>> a = [5 3 7 8 9 2 1 4]
```

```
>> b = [2 6 4 3 8 7 9 5]; % output display suppressed
```

To enter a matrix with real elements

```
>> A = [5 3 7; 8 9 2; 1 4.2 6e-2]
```

To enter a matrix with complex elements

```
>> X = [5+3j 7+8j; 9+2j 1+4j];
```

Transpose of a matrix

```
>> A_trans = A'
```

Determinant of a matrix

```
>> A_det = det(A)
```

Inverse of a matrix

```
>> A_inv = inv(A)
```

Matrix multiplication

```
>> C = A * A_trans
```

Array multiplication

```
>> c = a .* b      % a.*b denotes element-by-element multiplication.  
                  % vectors a and b must have the same dimensions.
```

## 2. Few useful commands

```
>>who      % lists variables in workspace (The MATLAB workspace  
consists of the variables you create and store in memory during a MATLAB session.)
```

```
>>whos      % lists variables and their sizes
```

```
>> help inv
```

The online help system is accessible using the help command. Help is available for functions e.g. help inv and for Punctuation help punct. A useful command to get started is intro, which covers the basic concepts in MATLAB language. Demonstration programs can be started with the command demo. Demos can also be invoked from the **Help Menu** at the top of the window.

```
>>lookfor inverse
```

The function *lookfor* becomes useful when one is not sure of the MATLAB function name.

```
>>clc      %clear command window
```

```
>> save session1      % workspace variables stored in session1.mat
```

```
>>save myfile.dat a b -ascii  % saves a,b in myfile.dat in ascii format
```

```
>>clear      % clear workspace
```

```
>>who
```

```
>>load session1      % To load variables to workspace
```

```
>>who
```

```
>>pwd      % present working directory
```

```
>>disp(' I have successfully completed MATLAB basics')
```

### More Matrix manipulations

$A = [1 \ 3 \ 5; 2 \ 4 \ 4; 0 \ 0 \ 3]$

$B = [1 \ 0 \ 1; 2 \ 2 \ 2; 3 \ 3 \ 3]$

Accessing a submatrix

$A(1:2,2:3); A(1,:); B(:,2)$

Concatenating two matrices

$D = [A \ B]; E = [A;B];$

Adding a row

$A(4,:) = [1 \ 1 \ 0]$

Deleting a column

$B(:,2) = []$

### Matrix generation functions

`zeros(3,3)` - 3x3 matrix of zeroes

`ones(2,2)` - 2x2 matrix of ones

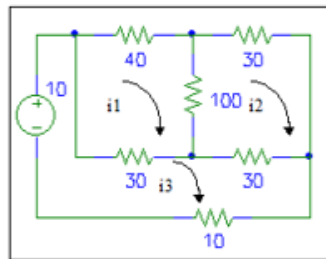
`eye(3)` - identity matrix of size 3

`rand(5)` - 5x5 matrix of random numbers

### ✓ Find the loop currents of the circuit given in Fig. Below..

Network equations are:

$$\begin{bmatrix} 170 & -100 & -30 \\ -100 & 160 & -30 \\ -30 & -30 & 70 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix}$$



### Sample Solution a:

(using command line editor)

```
>> Z = [170 -100 -30; -100 160 -30; -30 -30 70];
```

```
>> v = [0; 0; 10];
```

```
>> i = inv(Z) * v
```

### ✓ Creating script files

Editing a file: **Home Menu New -> Script**, invokes MATLAB Editor/Debugger.

Save file with extension **.m**

Sample Solution b: *(using ex\_1b.m file)*

Edit the following commands in the Editor and save as ex\_1b.m.

A line beginning with % sign is a comment line.

```
% ex_1b.m
clear; clc;

% Solution of Network equations
Z = [170 -100 -30; -100 160 -30; -30 -30 70];
v = [0; 0; 10];
disp('The mesh currents are : ')
i = inv(Z)*v
```

Typing ex\_1b at the command prompt will run the script file, and all the 7 commands will be executed sequentially. The script file can also be executed from the **Run** icon in the MATLAB Editor.

Sample Solution c: *(interactive data input using ex\_1c.m)*

```
% ex_1c.m
% Interactive data input and formatted output
clear; clc;

% Solution of Network equations
Z = input('Enter Z : ');
v = input('Enter v : ');
i = Z\ v; % Left division - computes inv(Z)*v
disp('The results are : ')
fprintf('i1 = %g A, i2 = %g A, i3 = %g A \n', i(1),i(2),i(3))
```

Results:

```
i =  0.1073
    0.1114
    0.2366
```

## Function File

A *function file* is also an m-file, just like a script file, except it has a function definition line at the top that defines the input and output explicitly.

*function*<output\_list> = *fname* <input\_list>

Save the file as *fname.m*. The filename will become the name of the new command for MATLAB. Variables inside a function are local. Use *global* declaration to share variables.

### ✓ *Create a function file for rectangular to polar conversion*

#### Sample solution:

```
function [r,theta] = r2p(x)
% r2p - function to convert from rectangular to polar
% Call syntax: [r,theta] = r2p(x)
% Input: x = complex number in the form a+jb
% Output: [r,theta] = the complex number in polar form

r = abs(x);
theta = angle(x)*180/pi;
```

Save the code in a file *r2p.m*.

Executing the function *r2p* in the Command Window

```
>> y = 3+4j;
>> [r,th] = r2p(y)
```

### ✓ *Write a function factorial to compute the factorial for any integer n.*

## Programming Tips

Writing efficient MATLAB code requires a programming style that generates small functions that are vectorised. Loops should be avoided. The primary way to avoid loops is to use toolbox functions as much as possible.

MATLAB functions operate on arrays just as easily as they operate on scalars. For example, if **x** is an array, then **cos(x)** returns an array of the same size as **x** containing the cosine of each element of **x**.

**Avoiding loops** - Since MATLAB is an interpreted language, certain common programming habits are intrinsically inefficient. The primary one is the use of *for* loops to perform simple operations over an entire matrix or vector. Wherever possible, you should try to find a vector function that will accomplish the desired result, rather than writing loops.

For example, to sum all the elements of a matrix

By using *for* loops:

```
[Nrows, Ncols]=size(x);  
xsum=0.0;  
for m=1:Nrows  
    for n=1:Ncols  
        xsum=xsum+x(m,n);  
    end;  
end;
```

Vectorised code:

```
xsum=sum(sum(x));
```

**Repeating rows or columns** - If the matrix has all same values use `ones(M,N)` and `zeros(M,N)`. To replicate a column vector `x` to create a matrix that has identical columns, `x=(12:-2:0)'`; `X= x*ones(1,11)`.

Experience has shown that MATLAB is easy to learn and use. You can learn a lot by exploring the Help & Demos provided and trying things out. One way to get started is by viewing and executing script files provided with the software. Don't try to learn everything at once. Just get going and as you need new capabilities find them, or, if necessary, make them yourself.



## **LAB NO.: 10**

# **MODULAR PROGRAMMING – RECURSIVE FUNCTIONS, STRUCTURES, POINTERS**

### **Objectives:**

In this lab, student will be able to:

1. To Learn the concept of recursion and to write recursive functions
2. Declare and initialize pointer variable
3. Write basic operations and programs using structures
4. Access a variable through its pointer

### **Introduction:**

- A recursive function is a function that invokes/calls itself directly or indirectly.
- A Pointer is a memory location or a variable which stores the address of another variable in memory
- A structure in the C programming language (and many derivatives) is a composite data type (or record) declaration that defines a physically grouped list of variables to be placed under one name in a block of memory.
- A file is a place on disc where group of related data is stored.

### **Steps to Design a Recursive Algorithm**

- **Base case:**
  - for a small value of  $n$ , it can be solved directly
- **Recursive case(s)**
  - Smaller versions of the same problem
- Algorithmic steps:
  - Identify the base case and provide a solution to it
  - Reduce the problem to smaller versions of itself
  - Move towards the base case using smaller versions

## Solved Exercise

Program to explain the concept of recursive functions

```
#include<stdio.h>
long factorial (long a) {
    if (a ==0) //base case
        return (1);
    return (a * factorial (a-1));
}
int main () {
    long number;
    printf("Please type a number: ");
    scanf("%d",&number);
    printf("%d factorial is %ld",number, factorial (number));
    return 0;
}
```

## Declaring and initializing pointers:

### Syntax:

```
data_type * pt_name;
```

This tells the compiler 3 things about the pt\_name:

- The asterisk (\*) tells the variable pt\_name is a pointer variable.
- pt\_name needs a memory location.
- pt\_name points to a variable of type data\_type

## Solved Exercise:

```
#include<stdio.h>
int main()
{
    int var1 = 11; //two integer variables
    int var2 = 22;
    int *ptr;      //pointer to integer
    ptr = &var1;   //pointer points to var1
}
```

```

        printf("%d",*ptr);    //print contents of pointer (11)
        ptr = &var2;         //pointer points to var2
        printf("%d",*ptr);    //print contents of pointer (22)
    return 0;
}

```

## **Declaration and initialization of structures:**

### **Déclaration :**

```

struct student
{
    int rollno, age;
    char name[20];
} s1, s2, s3;

```

### **Initialisation :**

```

int main( ){
    struct
    {
        int rollno;
        int age;
    } stud={20, 21};
    ...
    ...
    return 0;
}

```

### **Solved Exercise:**

```

#include<stdio.h>
struct Book{           //Structure Definition
    char title[20];
    char author[15];
    int pages;
    float price;
};

```

```

int main( ){
    struct Book b[10];
    int i,j;
    printf("Input values");
    for (i=0;i<10;i++){
        scanf("%s %s %d %f",b[i].title,b[i].author,&b[i].pages,&b[i].price);
    }
    for (j=0;j<10;j++){
        printf("title %s\n author %s\n pages %d\n price%f\n",b[j].title,b[j].author,b[j].pages,b[j].price);
    }
    return 0;
}

```

## Lab Exercises

With the knowledge of recursive functions, Structures and pointers

Write C programs as specified below:

1. Write a recursive function, **GCD** to find the GCD of two numbers. Write a main program which reads 2 numbers and finds the GCD of the numbers using the specified function. Ex: GCD of 9, 24 is 3.
2. Write a recursive function **FIB** to generate  $n^{\text{th}}$  Fibonacci term. Write a main program to print first N Fibonacci terms using function FIB.  
[Hint: Fibonacci series is 0, 1, 1, 2, 3, 5, 8 ...]
3. Find the maximum number in the input integer array using pointers.
4. Create a student record with name, rollno, marks of 3 subjects (m1, m2, m3). Compute the average of marks for 3 students and display the names of the students in ascending order of their average marks.

## Additional Exercises

1. Find the length of the string using pointers.
2. Write a program to multiply two numbers using a recursive function.

[Hint: Multiplication using repeated addition]

3. Write a C program to find the frequency of a word in an inputted string using pointers.
4. Create an employee record with emp-no, name, age, date-of-joining (year), and salary. If there is 20% hike on salary per annum, compute the retirement year of each employee and the salary at that time. [standard age of retirement is 55]

## LAB NO.: 11

# MATLAB GUI, ONLINE DOCUMENTATION WITH LIVE EDITOR – 2D and 3D plots

### Plot with Symbolic Plotting Functions

MATLAB<sup>®</sup> provides many techniques for plotting numerical data. Graphical capabilities of MATLAB include plotting tools, standard plotting functions, graphic manipulation and data exploration tools, and tools for printing and exporting graphics to standard formats. Symbolic Math Toolbox<sup>™</sup> expands these graphical capabilities and lets you plot symbolic functions using:

- fplot to create 2-D plots of symbolic expressions, equations, or functions in Cartesian coordinates.
- fplot3 to create 3-D parametric plots.
- ezpolar to create plots in polar coordinates.
- fsurf to create surface plots.
- fcontour to create contour plots.
- fmesh to create mesh plots.

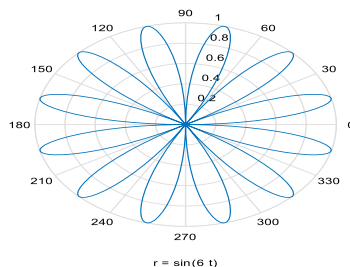
### Illustration

Plot a symbolic expression or function in polar coordinates  $r$  (radius) and  $\theta$  (polar angle) by using `ezpolar`. By default, `ezpolar` plots a symbolic expression or function over the interval  $0 < \theta < 2\pi$ .

Plot the symbolic expression  $\sin(6t)$  in polar coordinates.

```
>>syms t
```

```
>>ezpolar(sin(6*t))
```



2D Plot of  $\sin(6t)$

## Exercise – 1

- ✓ Plot the symbolic expression  $\sin(6x)$  by using fplot. By default, fplot uses the range  $-5 < x < 5$
- ✓ Plot the function  $\sin(x)$  between  $0 \leq x \leq 4\pi$ .
  - Hint: Use linspace function to create an x-array of 100 samples between 0 and  $4\pi$ .
  - The linspace function generates linearly spaced vectors.  
[>>x = linspace(0,4\*pi,100);]
- ✓ Plot of parametric space curve  $x(t) = t$ ,  $y(t) = t^2$ ,  $z(t) = t^3$ ;  $0 \leq t \leq 1$ , using plot3 and give suitable labels.

## Matlab App Designer

App Designer is a rich drag-and-drop environment introduced in R2016a, and it is the recommended environment for building most apps. It includes a fully integrated version of the MATLAB editor. The layout and code views are tightly linked so that changes you make in one view immediately effects the other. A larger set of interactive controls is available, including gauges, lamps, knobs, and switches. Most graphics functionalities are supported in App designer.

Use MATLAB Functions to Create Apps Programmatically You can also code the layout and behavior of your app entirely using MATLAB functions. In this approach, you create a traditional figure and place interactive components in that figure programmatically. These apps support the same types of graphics and interactive components that GUIDE supports, as well as tabbed panels.

## Example illustration

Creating waveform generator with three push buttons configured to generate SIN, COS and TAN waveforms with vector space  $x = \text{linspace}(0, 4\pi, 100)$ ;

Write Matlab scripts to generate waveform on button click event in callback code and publish the project as

- Matlab App
- Web App
- Standalone Desktop App

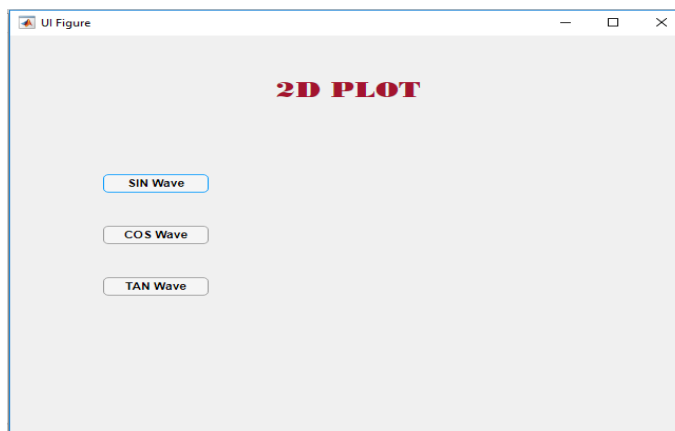
Procedure to create App designer

- Click on App under Home tab -> New
- Drag and drop controls
- Label could be used for title
- Push button for instantiating an event based on RUN and click

- Change configuration properties of the label, push button etc. to change font, size, alignment, background color etc.
- Click on callback button to write code for specific task
 

```
% Button pushed function: SINWaveButton
function SINWaveButtonPushed(app, event)
    x=linspace(0,4*pi,100);
    y=sin(x);
    plot(y)
end
```
- Run the function to get the output as desired waveform
- Click on share tab to create Matlab App, Web App and Standalone Desktop App of this project

### Design view

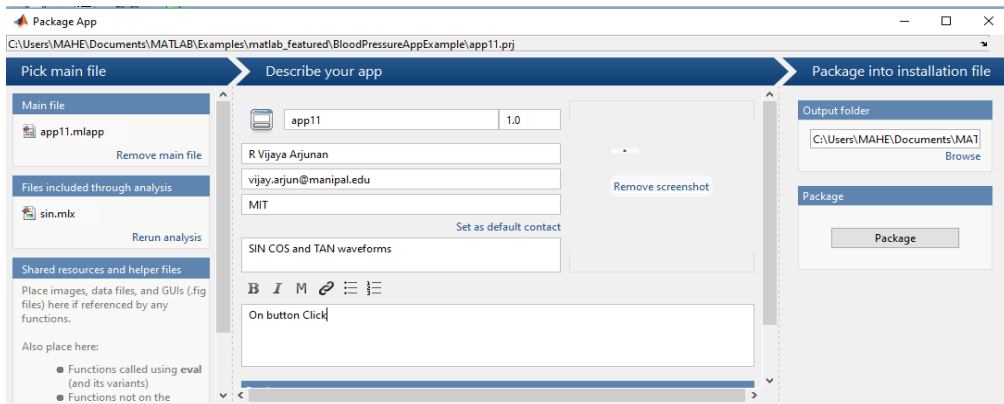


### Code view

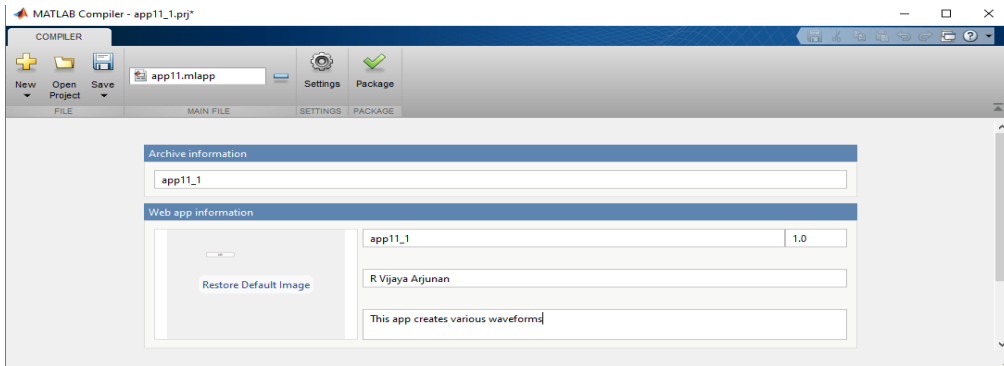
```
% Button pushed function: SINWaveButton
function SINWaveButtonPushed(app, event)
    x=linspace(0,4*pi,100);
    y=sin(x);
    plot(y)
end
```



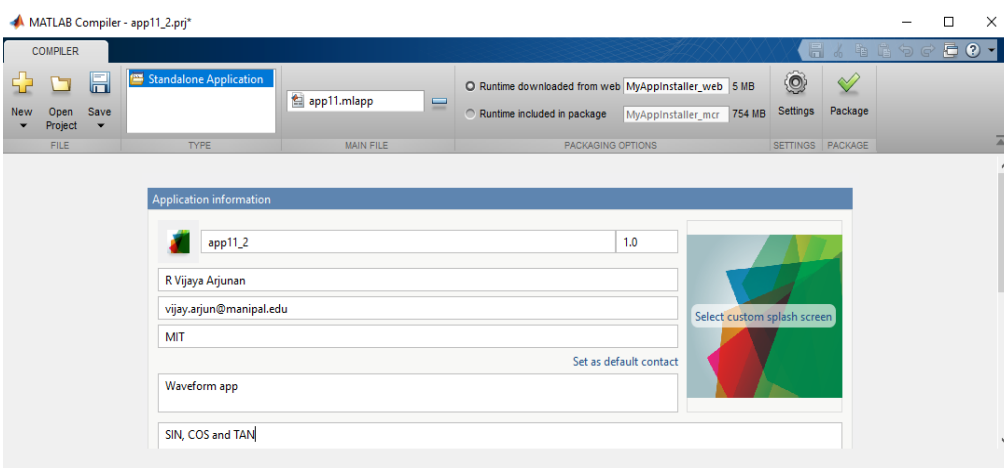
## Web sharing Matlab App



## Web App

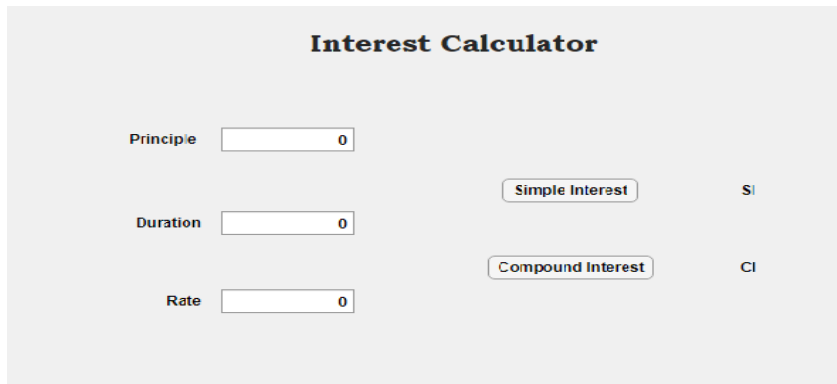


## Standalone App



## Exercise – 2

Create a following GUI App to find and display the Simple and Compound interest for the given principle, duration and rate in respective text (numeric) fields on click of push button. Refer below GUI for your representation.



The image shows a GUI titled "Interest Calculator". It has three input fields on the left: "Principle" with a value of 0, "Duration" with a value of 0, and "Rate" with a value of 0. On the right, there are two buttons: "Simple Interest" and "Compound Interest". To the right of these buttons are labels "SI" and "CI" respectively.

## Exercise – 3

Create a GUI app find below using appropriate controls.

### Perimeter and Area of the Square:

- Perimeter of square =  $4 \times S$ .
- Area of square =  $S \times S$ .
- Diagonal of square =  $S\sqrt{2}$ ; ( $S$  is the side of square)

### Perimeter and Area of the Triangle:

- Perimeter of triangle =  $(a + b + c)$ ; ( $a, b, c$  are 3 sides of a triangle)
- Area of triangle =  $\sqrt{s(s - a)(s - b)(s - c)}$ ; ( $s$  is the semi-perimeter of triangle)
- $S = 1/2 (a + b + c)$
- Area of triangle =  $1/2 \times b \times h$ ; ( $b$  base,  $h$  height)
- Area of an equilateral triangle =  $(a^2\sqrt{3})/4$ ; ( $a$  is the side of triangle)

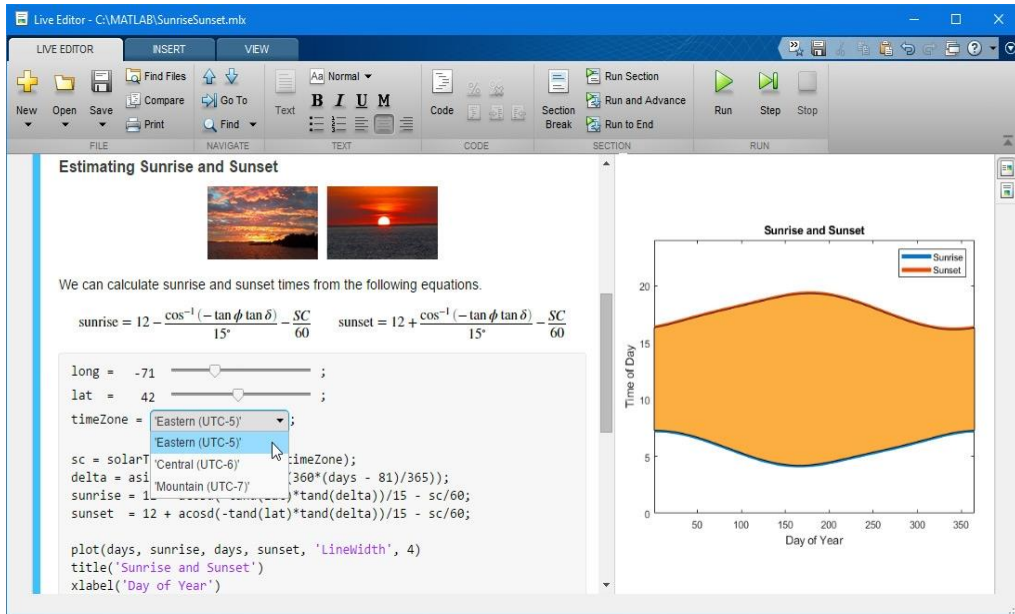
## Live editor

### Turn Your Code into an Interactive Document

Enhance your code and output with formatting, images, and hyperlinks to turn your live script into a story. Insert equations using the interactive editor or create them using LaTeX. Add interactive controls to set values in your script. You can then share your live scripts directly with colleagues so they can reproduce or expand on your work, or create static PDF, HTML, and LaTeX documents for publication.

## Create Live Functions

Create and debug live functions and scripts. Add formatted documentation to your functions.



## Illustration to create live editor [.mlx]

*Plot the function  $\sin(x)$  between  $0 \leq x \leq 4\pi$  and save this file as .pdf and publish. Add required steps as comments for a detailed narration.*

step 1:

Create an x-array of 100 samples between 0 and  $4\pi$ .

```
x=linspace(0,4*pi,100);
```

step 2:

Calculate  $\sin(\cdot)$  of the x-array

```
y=sin(x);
```

step 3:

Plot the y-array

```
plot(y)
```

grid on

```
title('SIN plot')
```

```
xlabel('time period')
```

```
ylabel('hhhh')
```

Note:

- Click on the runtime output to access controls like “grid”, “title”, “x label”, “y label” etc on “Figure” to add code and update
- Click on save as control in Home tab to save the desired document in PDF.

#### Exercise – 4

Use Matlab **Live Editor** illustrative procedure to generate PDF documents of following assignments.

- Plot of parametric space curve  $x(t) = t$ ,  $y(t) = t^2$ ,  $z(t) = t^3$ ;  $0 \leq t \leq 1$ , using ***plot3*** and give suitable labels through “figure” tab and update the code in program code.
- 2D plot for the function,  $x=t$ ,  $y=e^t$ ,  $0 \leq t \leq 2\pi$  using ***semilogy***.
- Both 3D-mesh and 3D-colored surface in the **same Figure window** using **subplot**

[Hint: **subplot (221)**, **mesh(Z)**]

## LAB NO.: 12

# ENGINEERING APPLICATIONS WITH MATLAB SIMULINK

### *Objectives:*

Analysis of simple systems using SIMULINK

Simulink<sup>®</sup> is a block diagram environment for multi-domain simulation and Model-Based Design. It supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. In addition, it is integrated with MATLAB<sup>®</sup>, enabling you to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.

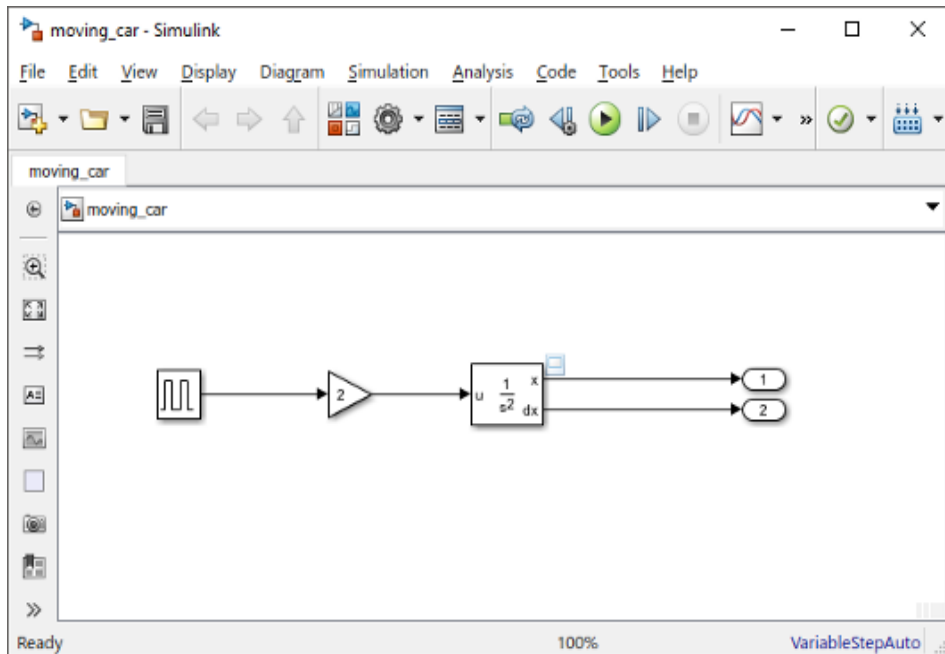
You can use Simulink<sup>®</sup> to model a system and then simulate the dynamic behavior of that system. The basic techniques you use to create a simple model in this tutorial are the same as those you use for more models that are complex.

### **Motion of a car – Simulink model – Illustration tutorial**

A car is typically in motion WHILE the pedal is pressed. It idles AFTER and comes to a stop. After a brief press of the accelerator pedal.

A Simulink block is a model element that defines a mathematical relationship between its input and output. To create this simple model, you need four Simulink blocks.


Block Name	Block Purpose	Model Purpose
Pulse Generator	Generate an input signal for the model	Represent the accelerator pedal
Gain	Multiply the input signal by a factor	Calculate how pressing the accelerator affects the car acceleration
Integrator, Second-Order	Integrate input signal twice	Obtain position from acceleration
Out port	Designate a signal as an output from the model	Designate the position as an output from the model

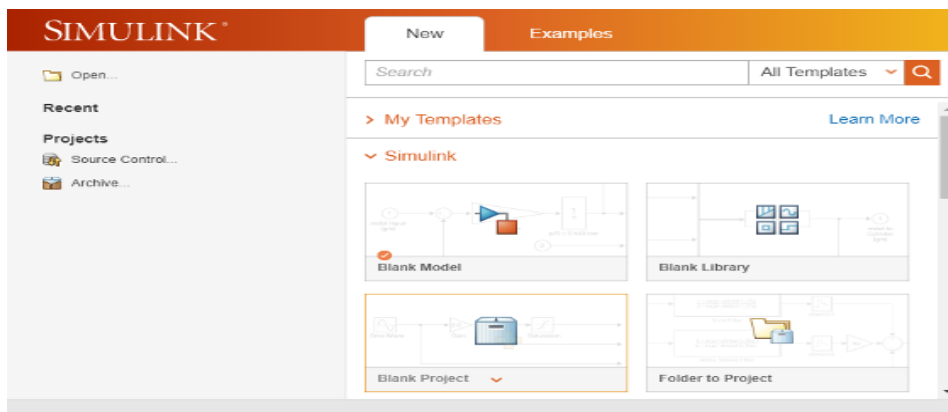


Simulating this model integrates a brief pulse twice to get a ramp. The results display in a Scope window. The input pulse represents a press of the gas pedal — 1 when the pedal is pressed and 0 when it is not. The output ramp is the increasing distance from the starting point.

## Open New Model

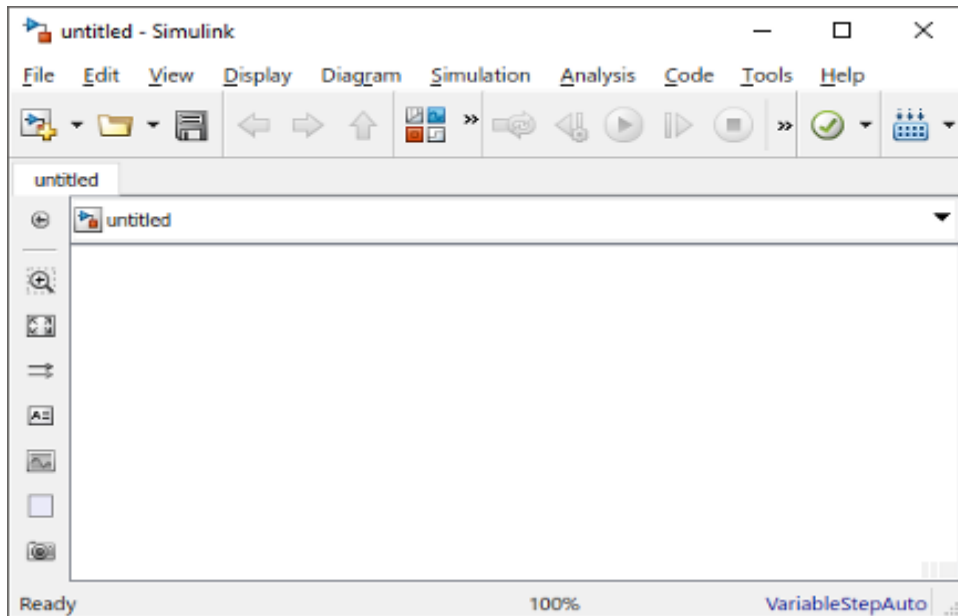
Use the Simulink Editor to build your models.

1. Start MATLAB®. From the MATLAB tool-strip, click the **Simulink** button 



2. Click the **Blank Model** template.


The Simulink Editor opens.

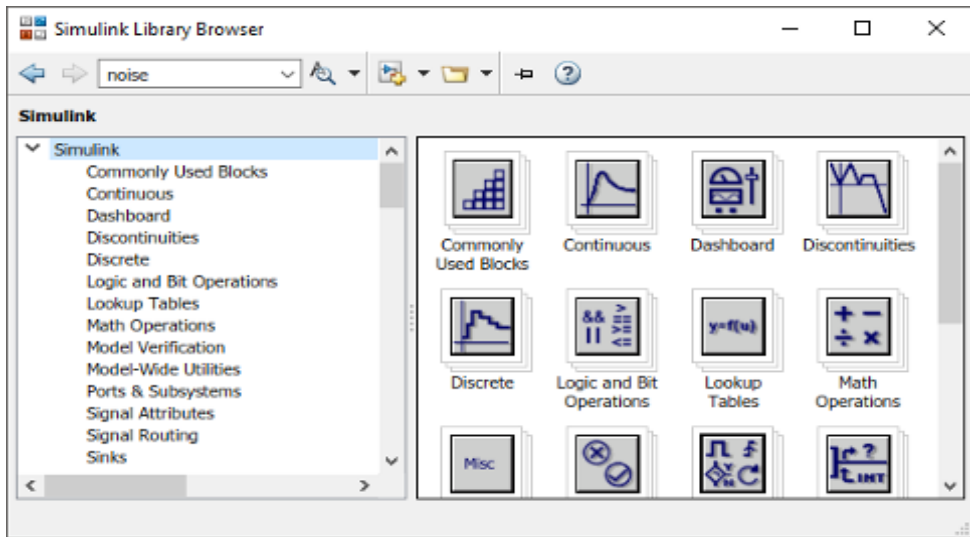



3. From the **File** menu, select **Save as**. In the **File name** text box, enter a name for your model, For example, `simple model`. Click **Save**. The model is saved with the file extension `.slx`.

### Open Simulink Library Browser

Simulink provides a set of block libraries, organized by functionality in the Library Browser. The following libraries are common to most workflows:

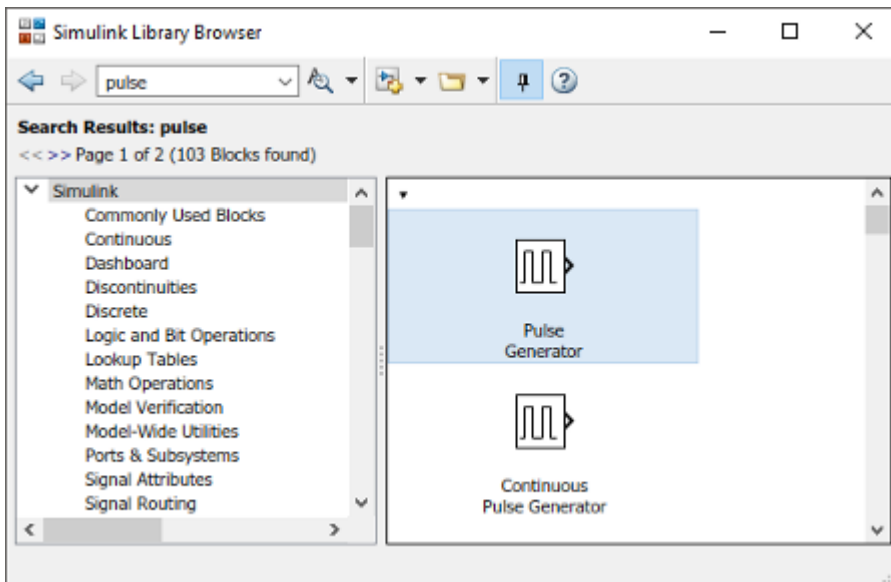
- Continuous — Blocks for systems with continuous states
- Discrete — Blocks for systems with discrete states
- Math Operations — Blocks that implement algebraic and logical equations
- Sinks — Blocks that store and show the signals that connect to them
- Sources — Blocks that generate the signal values that drive the model
- From the Simulink Editor toolbar, click the **Library Browser** button 



- Set the Library Browser to stay on top of the other desktop windows. On the Library Browser toolbar, select the **Stay on top** button 

To browse through the block libraries, select a category and then a functional area in the left pane. To search all of the available block libraries, enter a search term.

For example, find the Pulse Generator block. In the search box on the browser toolbar, enter pulse, and then press the Enter key. Simulink searches the libraries for blocks with pulse in their name or description, and then displays the blocks.





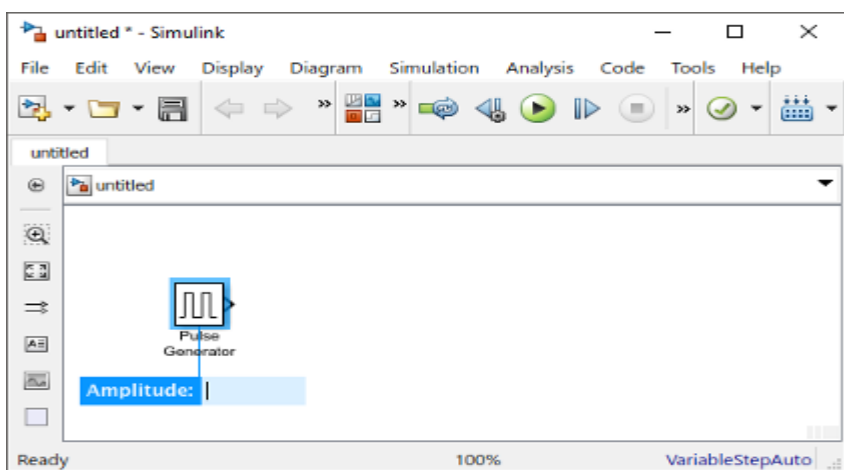
Get detailed information about a block. Right-click a block and then select **Help for the Pulse Generator block**. The Help browser opens with the reference page for the block.

Blocks typically have several parameters. You can access all parameters by double-clicking the block.

### Add Blocks to a Model

To start building the model, browse the library and add the blocks.

- From the Sources library, drag the Pulse Generator block to the Simulink Editor. A copy of the Pulse Generator block appears in your model with a text box for the value of the **Amplitude** parameter. Enter 1.



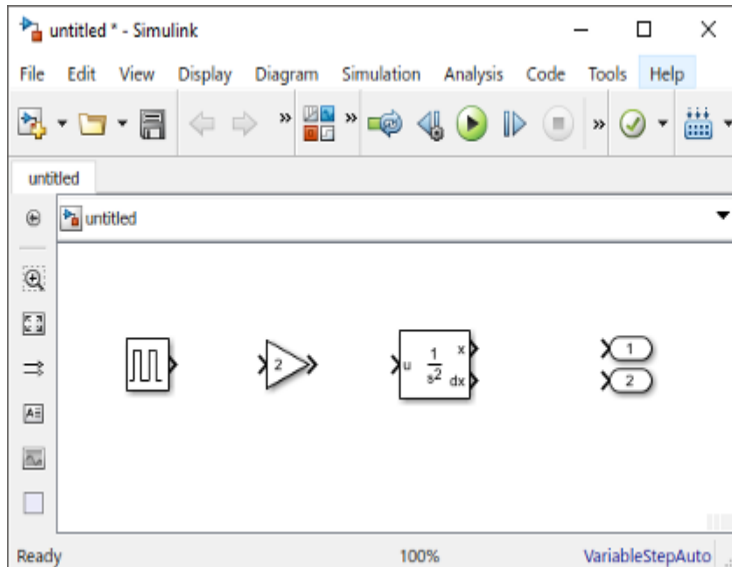
Parameter values are held throughout the simulation.

- Add the following blocks to your model using the same approach.

Block	Library	Parameter
Gain	Simulink/Math Operations	Gain: 2
Integrator, Second Order	Simulink/Continuous	Initial condition: 0
Out port	Simulink/Sinks	Port number: 1

- Add a second out port block by copying the existing one and pasting it at another point using keyboard shortcuts.
- Your model now has the blocks you need.

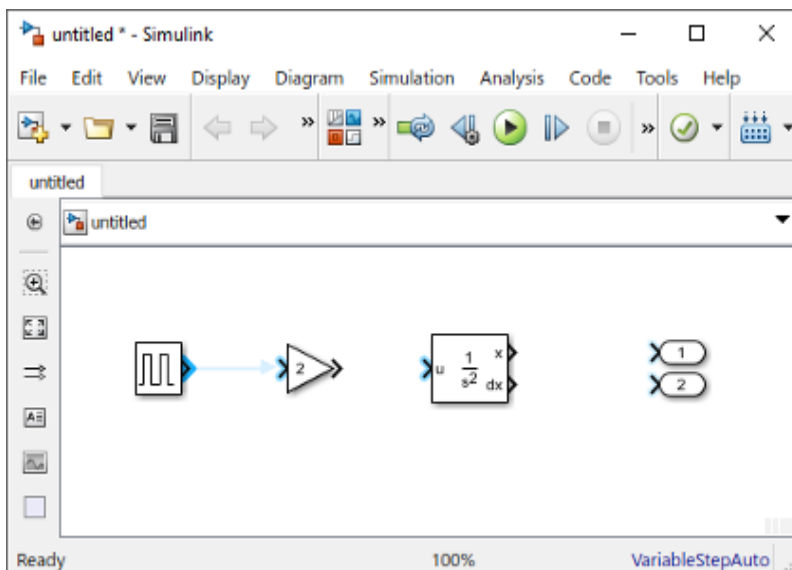
- Arrange the blocks as follows by clicking and dragging each block. To resize a block, click and drag a corner.



## Connect Blocks

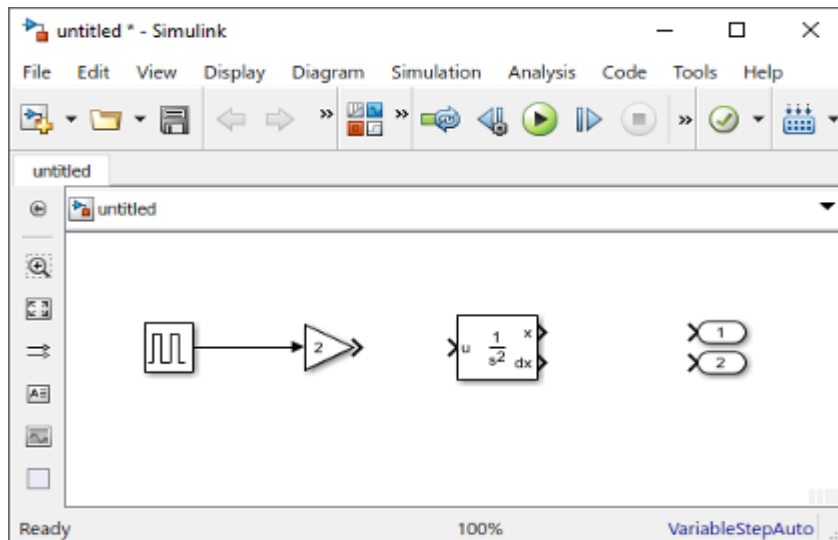
Connect the blocks by creating lines between output ports and input ports.

4. Click the output port on the right side of the Pulse Generator block.  
The output port and all input ports suitable for a connection get highlighted.

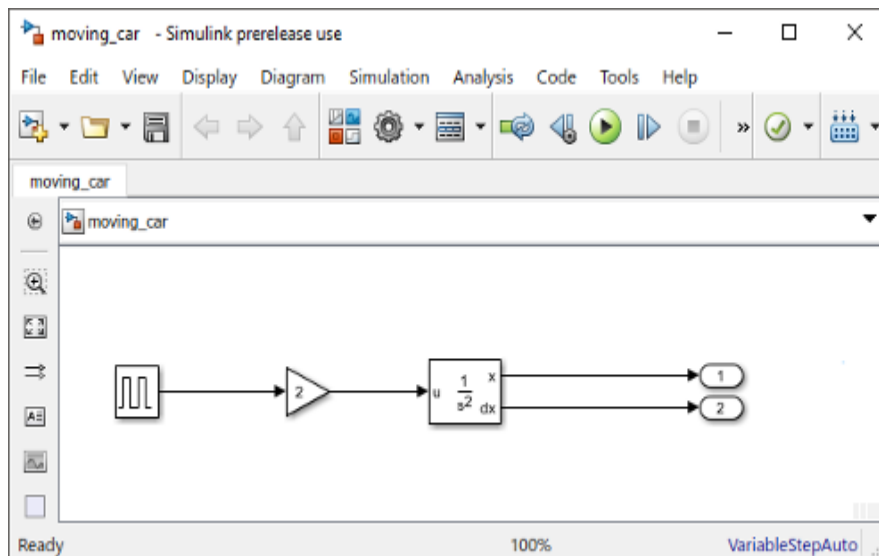


- Click the input port of the Gain block.

Simulink connects the blocks with a line and an arrow indicating the direction of signal flow.



- Connect the output port of the Gain block to the input port on the Integrator, Second Order block.
- Connect the two outputs of the Integrator, Second Order block to the two Output blocks.
- Save your model. Select **File > Save** and provide a name.



## Add Signal Viewer

To view simulation results, connect the first output to a Signal Viewer.

Access the context menu by right clicking the signal. Select **Create & Connect Viewer** > **Simulink** > **Scope**. A viewer icon appears on the signal and a scope window opens.



You can open the scope at any time by double-clicking the icon.


## Run Simulation

After you define the configuration parameters, you are ready to simulate your model.

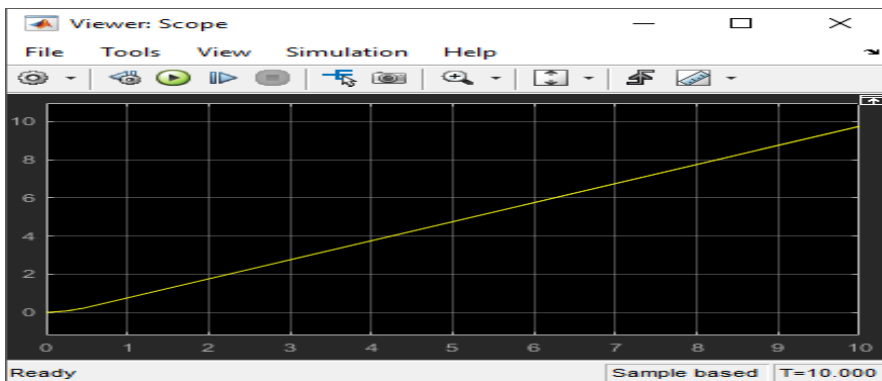
- On the model window, set the simulation stop time by changing the value at the toolbar.



The default stop time of 10.0 is appropriate for this model. This time value has no unit. Time unit in Simulink depends on how the equations are constructed. This example simulates the simplified motion of a car for 10 seconds — other models could have time units in milliseconds or years.

- To run the simulation, click the **Run** button .

The simulation runs and produces the output in the viewer.

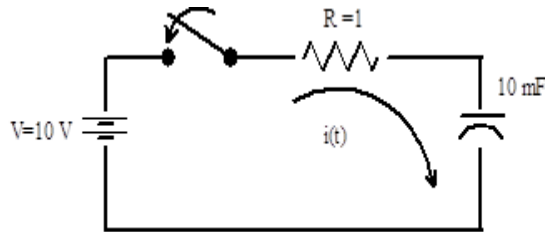


### Exercises:

#### Verify the following Electrical circuit output using Simulink

1. An RC series circuit with  $R = 1\Omega$ , &  $C = 10\text{mF}$  is connected to a dc source of 10V through a switch. Plot the applied voltage, current and the capacitor voltage for time,  $0 \leq t \leq 2\text{s}$ , if the switch is closed at  $t = 1\text{s}$  & the circuit elements are initially relaxed.

Circuit



Differential equation defining above circuit

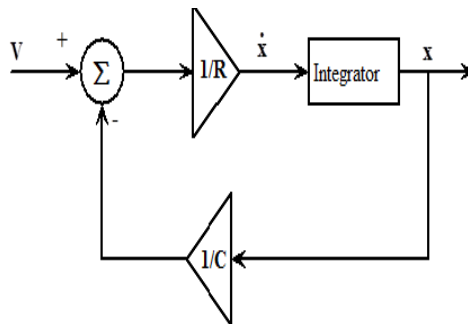
$$V(t) = R i(t) + \frac{1}{C} \int i(t) dt$$

Taking  $x = \int i(t) dt$

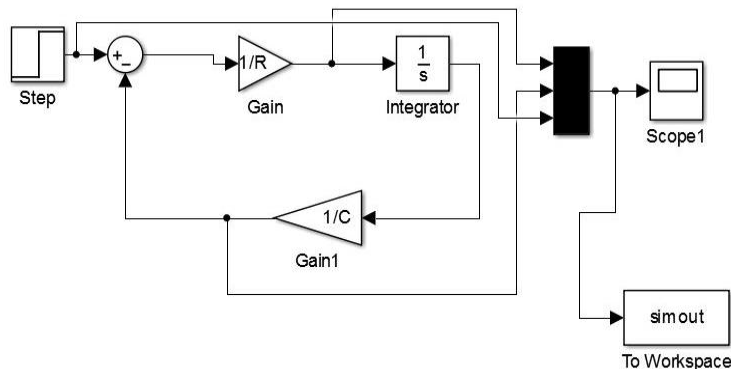
The above equation may be re-written as

$$R \dot{x} = V(t) - \frac{1}{C} x$$

Block diagram describing the above equation is



Simulink model describing the above block diagram is



- To invoke SIMULINK from MATLAB Command Window

In Home menu, Select New - Simulink Model

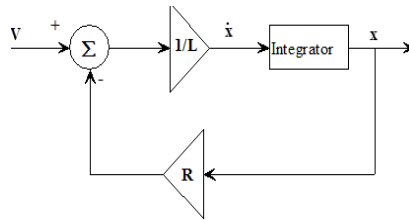
- Draw the block schematic as shown.
  - Initialize each block by setting appropriate values.
  - Step function applied at  $t=1$ .
- Set simulation parameters (Start time=0, Stop time =2, Solver Options: Type – Variable step, ode45, Max. step size = 0.01).
    - Run the simulation using the icon provided.
    - Observe the result using Scope.
    - Plot results in MATLAB using plot command.
2. An RL series circuit with  $R = 2 \Omega$ , &  $L = 0.5 \text{ H}$  is connected to a dc source of 10V through a switch. Plot the applied voltage, current and the capacitor voltage for time,  $0 \leq t \leq 5\text{s}$ , if the switch is closed at  $t = 1\text{s}$  & the circuit elements are initially relaxed.

**Equation**

$$V(t) = L \frac{di(t)}{dt} + Ri(t) \quad (\text{let } x = i(t), \text{ hence } \dot{x} = \frac{di(t)}{dt})$$

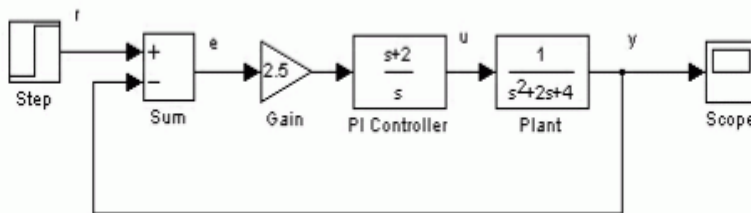
$$L \frac{di(t)}{dt} = V(t) - Ri(t)$$

## Block diagram



## Additional exercises:

- Simulate the following model and analyse the signal



Hint: Change the numerator to [1 2] and the denominator to [1 0] for PI controller.  
Change the denominator to [1 2 1] for Plant.

## REFERENCES

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming language (2e)*, Pearson Education, 2015.
2. Deital.P. J and Deitel.H.M, *C: How to program (7e)*, Pearson Education, 2010.
3. Balagurusamy.E, *Computing fundamentals and C programming (1e)*, MC GRAW HILL INDIA, 2017.
4. Delores Etter, *Introduction to MATLAB*, Pearson Education India, 2019.
5. Stormy Attaway, *Matlab: A practical Introduction to Programming and Problem Solving (4e)*, Butterworth-Heinemann, Elsevier, 2017.

## C LANGUAGE QUICK REFERENCE

### PREPROCESSOR

```
// Comment to end of line
/* Multi-line comment */

#include <stdio.h>    // Insert standard header file
#include "myfile.h"   // Insert file in current directory
#define X some text   // Replace X with some text
#define F(a,b) a+b     // Replace F(1,2) with 1+2
#define X \
    some text          // Line continuation
#undef X              // Remove definition
#ifdef X              // Conditional compilation (#ifdef X)
#else                 // Optional (#ifndef X or #if !defined(X))
#endif                // Required after #if, #ifdef
```

### LITERALS

```
255, 0377, 0xff      // Integers (decimal, octal, hex)
2147463647L, 0x7fffffffL // Long (32-bit) integers
123.0, 1.23e2         // double (real) numbers
'a', '\141', '\x61'   // Character (literal, octal, hex)
'\n', '\\', '\'', '\"', // Newline, backslash, single quote, double quote
"string\n"            // Array of characters ending with newline and \0
"hello" "world"       // Concatenated strings
true, false           // bool constants 1 and 0
```

### DECLARATIONS

```
int x;                // Declare x to be an integer (value undefined)
int x=255;             // Declare and initialize x to 255
short s; long l;       // Usually 16 or 32 bit integer (int may be either)
char c= 'a';           // Usually 8 bit character
unsigned char u=255; signed char m=-1; // char might be either
unsigned long x=0xffffffffL; // short, int, long are signed
float f; double d;     // Single or double precision real (never unsigned)
```



bool b=true;	// true or false, may also use int (1 or 0)
int a, b, c;	// Multiple declarations
int a[10];	// Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};	// Initialized array (or a[3]={0,1,2}; )
int a[2][3]={ {1,2,3},{4,5,6}};	// Array of array of ints
char s[]="hello";	// String (6 elements including '\0')
int* p;	// p is a pointer to (address of) int
char* s="hello";	// s points to unnamed array containing "hello"
void* p=NULL;	// Address of untyped memory (NULL is 0)
int& r=x;	// r is a reference to (alias of) int x
enum weekend {SAT, SUN};	// weekend is a type with values SAT and SUN
enum weekend day;	// day is a variable of type weekend
enum weekend {SAT=0,SUN=1};	// Explicit representation as int
enum {SAT,SUN} day;	// Anonymous enum
typedef String char*;	// String s; means char* s;
const int c=3;	// Constants must be initialized, cannot assign
const int* p=a;	// Contents of p (elements of a) are constant
int* const p=a;	// p (but not contents) are constant
const int* const p=a;	// Both p and its contents are constant
const int& cr=x;	// cr cannot be assigned to change x

## **STORAGE CLASSES**

int x;	// Auto (memory exists only while in scope)
static int x;	// Global lifetime even if local scope
extern int x;	// Information only, declared elsewhere

## **STATEMENTS**

x=y;	// Every expression is a statement
int x;	// Declarations are statements
;	// Empty statement
{	// A block is a single statement
int x;	// Scope of x is from declaration to end of block
a;	// In C, declarations must precede statements
}	
if (x) a;	// If x is true (not 0), evaluate a

else if (y) b;	// If not x and y (optional, may be repeated)
else c;	// If not x and not y (optional)
while (x) a;	// Repeat 0 or more times while x is true
for (x; y; z) a;	// Equivalent to: x; while(y) {a; z;}
do a; while (x);	// Equivalent to: a; while(x) a;
switch (x) {	// x must be int
case X1: a;	// If x == X1 (must be a const), jump here
case X2: b;	// Else if x == X2, jump here
default: c;	// Else jump here (optional)
}	
break;	// Jump out of while, do, for loop, or switch
continue;	// Jump to bottom of while, do, or for loop
return x;	// Return x from function to caller
try { a; }	
catch (T t) { b; }	// If a throws T, then jump here
catch (...) { c; }	// If a throws something else, jump here

## **FUNCTIONS**

int f(int x, int);	// f is a function taking 2 ints and returning int
void f();	// f is a procedure taking no arguments
void f(int a=0);	// f() is equivalent to f(0)
f();	// Default return type is int
inline f();	// Optimize for speed
f( ) { statements; }	// Function definition (must be global)

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

```
int main() { statements... }      or
int main(int argc, char* argv[]) { statements... }
```

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

## **EXPRESSIONS**

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation which is undefined. There are no runtime checks for arrays out of bounds, invalid pointers etc.

::X	// Global name X
t.x	// Member x of struct or class t
p → x	// Member x of struct or class pointed to by p
a[i]	// i'th element of array a
f(x, y)	// Call to function f with arguments x and y
x++	// Add 1 to x, evaluates to original x (postfix)
x--	// Subtract 1 from x, evaluates to original x
sizeof x	// Number of bytes used to represent object x
sizeof(T)	// Number of bytes to represent type T
++x	// Add 1 to x, evaluates to new value (prefix)
--x	// Subtract 1 from x, evaluates to new value
~x	// Bitwise complement of x
!x	// true if x is 0, else false (1 or 0 in C)
-x	// Unary minus
+x	// Unary plus (default)
&x	// Address of x
*p	// Contents of address p (*&x equals x)
x * y	// Multiply
x / y	// Divide (integers round toward 0)
x % y	// Modulo (result has sign of x)
x + y	// Add, or &x[y]
x - y	// Subtract, or number of elements from *x to *y
x << y	// x shifted y bits to left (x * pow(2, y))
x >> y	// x shifted y bits to right (x / pow(2, y))
x < y	// Less than
x <= y	// Less than or equal to
x > y	// Greater than
x >= y	// Greater than or equal to

<code>x == y</code>	// Equals
<code>x != y</code>	// Not equals
<code>x &amp; y</code>	// Bitwise and (3 & 6 is 2)
<code>x ^ y</code>	// Bitwise exclusive or (3 ^ 6 is 5)
<code>x   y</code>	// Bitwise or (3   6 is 7)
<code>x &amp;&amp; y</code>	// x and then y (evaluates y only if x (not 0))
<code>x    r</code>	// x or else y (evaluates y only if x is false(0))
<code>x = y</code>	// Assign y to x, returns new value of x
<code>x += y</code>	// x = x + y, also -= *= /= <<= >>= &=  = ^=
<code>x ? y : z</code>	// y if x is true (nonzero), else z
<code>x, y</code>	// evaluates x and y, returns y (seldom used)

### **STRING (Variable sized character array)**

<code>string s1, s2= "hello";</code>	//Create strings
<code>s1.size(), s2.size();</code>	// Number of characters: 0, 5
<code>s1 += s2 + ' ' + "world";</code>	// Concatenation
<code>s1 == "hello world";</code>	// Comparison, also <, >, !=, etc.
<code>s1[0];</code>	// 'h'
<code>s1.substr(m, n);</code>	// Substring of size n starting at s1[m]
<code>s1.c_str();</code>	// Convert to const char*
<code>getline(cin, s);</code>	// Read line ending in '\n'
<code>asin(x); acos(x); atan(x);</code>	// Inverses
<code>atan2(y, x);</code>	// atan(y/x)
<code>sinh(x); cosh(x); tanh(x);</code>	// Hyperbolic
<code>exp(x); log(x); log10(x);</code>	// e to the x, log base e, log base 10
<code>pow(x, y); sqrt(x);</code>	// x to the y, square root
<code>ceil(x); floor(x);</code>	// Round up or down (as a double)
<code>fabs(x); fmod(x, y);</code>	// Absolute value, x mod y

---