# MTE201 C Programming Basics

# COURSE SYNOPSES

C Programming: Introductory concepts, C fundamentals, operators and expression, data input and output, preparing and running a complete C program, control statements, functions, program structure, arrays, pointers, structures and unions, data files and low level programming. Advanced C Programming: Control statements, functions, program structure, arrays, pointers, structures and unions, data files and low level programming.

# 1.0. History of C

The C programming language was pioneered by Dennis Ritchie at AT&T Bell Laboratories in the early 1972. The language was formalized in 1988 by the American National Standard Institute (ANSI). It was not until the late 1970s, that this programming language began to gain widespread popularity and support. This was because until that time C compilers user re not readily available for commercial use outside of Bell Laboratories. C is a general-purpose programming language, and is used for writing programs in many different domains, such as operating systems, numerical computing, graphical applications, etc. It is a small language, with just 32 keywords. It provides "high-level" structured programming constructs such as statement grouping, decision making, and looping, as user ll as "low level" capabilities such as the ability to manipulate bytes and addresses.

## 1.1. Programming

Computers are really very dumb machines indeed because they do as instruct by the user. To solve a problem using a computer, you must express the solution to the problem in terms of the instructions of the particular computer. A computer program is just a collection of the instructions necessary to solve a specific problem. The approach or method that is used to solve the problem is known as an algorithm. Normally, to develop a program to solve a particular problem, you first express the solution to the problem in terms of an algorithm and then develop a program that implements that algorithm. So, the algorithm for solving the even/odd problem might be expressed as follows: First, divide the number by two. If the remainder of the division is zero, the number is even; otherwise, the number is odd. With the algorithm in hand, you can then proceed to write the instructions necessary to implement the algorithm on a particular computer system. These instructions would be expressed in the statements of a particular computer language, such as Visual Basic, Java, C++, or C.

## 1.2. Operating Systems

An operating system is a program that controls the entire operation of a computer system. All input and output (that is, I/O) operations that are performed on a computer system are channeled through the operating system. One of the most popular operating systems today is the Unix operating system, which was developed at Bell Laboratories. Unix is a rather unique operating system in that it can be found on many different types of computer systems, and in different "flavors," such as Linux or Mac OS X.

## 1.3. The C Compiler

A compiler analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on your particular computer system. The source code written in source file is the human readable source for your program. It needs to be "compiled" into machine language so that your CPU can actually execute the program as per the instructions given. The compiler compiles the source codes into final executable programs. The most frequently used and free available compiler is the GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective operating systems.

## 1.4. Integrated Development Environments (IDE)

This process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows you to easily manage large software programs, edit files in windows, and compile, link, run, and debug your programs.

## 1.5. Touring the Code::Blocks workspace

Figure 1 illustrates the Code::Blocks workspace, which is the official name of the massive mosaic of windows you see on the screen. The details in Figure 1 are rather small, but what you need to find are the main areas, which are called out in the figure:

1. Toolbars: These messy strips, adorned with various command buttons, cling to the top of the Code::Blocks window. There are eight toolbars, which you can rearrange, show, or hide. Don't mess with them until you get comfy with the interface.
2. Management: The window on the left side of the workspace features four tabs, though you may not see all four at one time. The window provides a handy oversight of your programming endeavors.
3. Status bar: At the bottom of the screen, you see information about the project and editor and about other activities that take place in Code::Blocks .
4. Editor: The big window in the center-right area of the screen is where you type code.

5. Logs: The bottom of the screen features a window with many, many tabs. Each tab displays information about your programming projects. The tab you use most often is named Build Log.

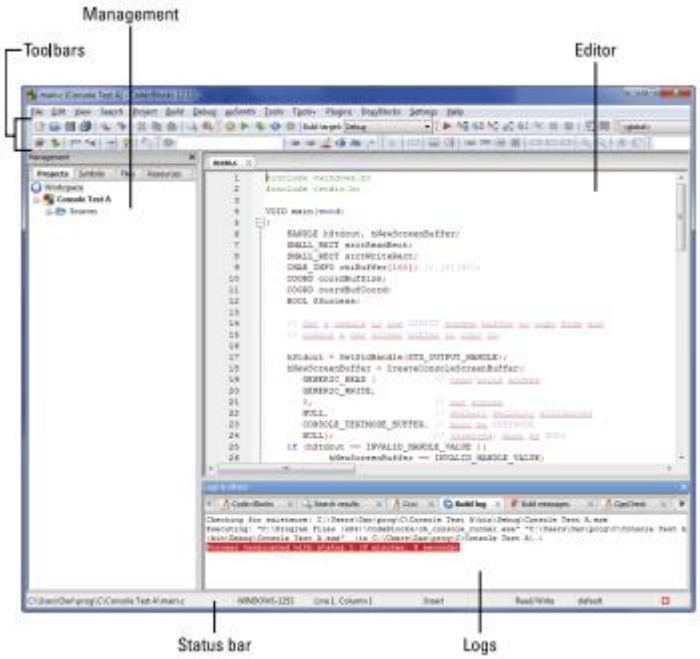



Figure 1 The Code::Blocks workspace.

## 1.6. Creating a new project

1. Start Code::Blocks . You see the Start Here screen, which displays the Code::Blocks logo and a few links. If you don't see the Start Here screen, choose File➪Close workspace.
2. Click the Create a New Project link. The New from Template dialog box appears, as shown in Figure 2.
3. Choose Console Application and then click the Go button. The Console Application Wizard appears. You can place a check mark by the item Skip This Page Next Time to skip over the wizard's first screen.
4. Click the Next button.
5. Choose C as the language you want to use, and then click the Next button. C is quite different from C++ — you can do things in one language that aren't allouser d in the other.

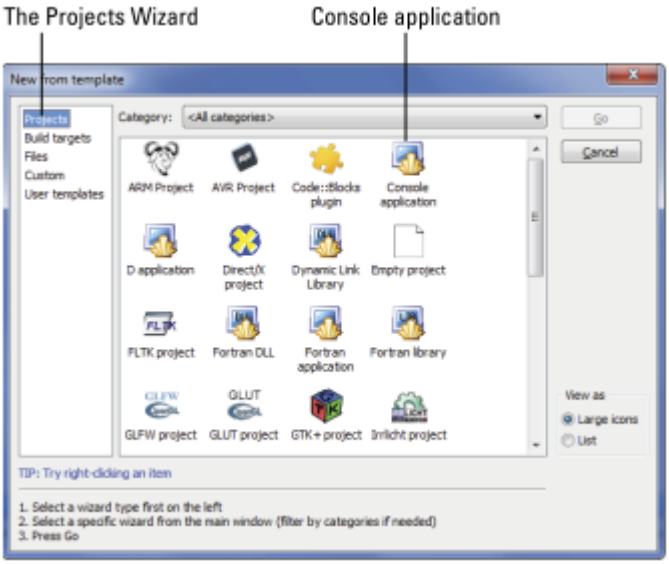



Figure 2 New projects

6. Type MTE201_01 as the project title. When you set the project title, the project's filename is automatically filled in.
7. Click the ... (Browse) button to the right of the text box titled Folder to Create Project In. I recommend that you create and use a special folder for all projects in this book.
8. Use the Make New Folder button in the Browse for Folder dialog box to create a project folder.
9. Click the OK button to select the folder and close the dialog box.
10. Click the Next button. The next screen (the last one) allows you to select a compiler and choose whether to create Debug or Release versions of your code, or both. The compiler selection is fine; the GNU GCC Compiler (or whatever is shown in the window) is the one you want.
11. Remove the check mark by Create Debug Configuration. You create this configuration only when you need to debug, or fix, a programming predicament that puzzles you.
12. Click the Finish button.

## 1.7. Features of C language

1. High level language: it is written in user understandable language making it user friendly and easy to comprehend.
2. Structured language: it improves clarity, quality and it reduces the develop time for designing a programming software

3. Rich library: it has its own library which includes most of the arithmetic and logic operation which are predefined. The user only includes the needed library in the code and their functionality can be executed without having to code them separately.
4. Extensibility: programs written in C language are highly extensible.
5. Recursion: this prevent the writing of the same function multiply times. Instead whenever the user needs the function user just have to call it. This help to reduce the time involved in the development cycle and also improves the code functionality.
6. Pointers: using the pointers user can directly interact with the physical memory of the computer system.
7. Faster execution: program execution is faster in C language than its predecessors.
8. Memory management: C language offers many functions where user can dynamically and directly interact with the memory of the computer system.

### 1.7.1. C-Tokens

Figure 3 presents the C-tokens in C programming language.

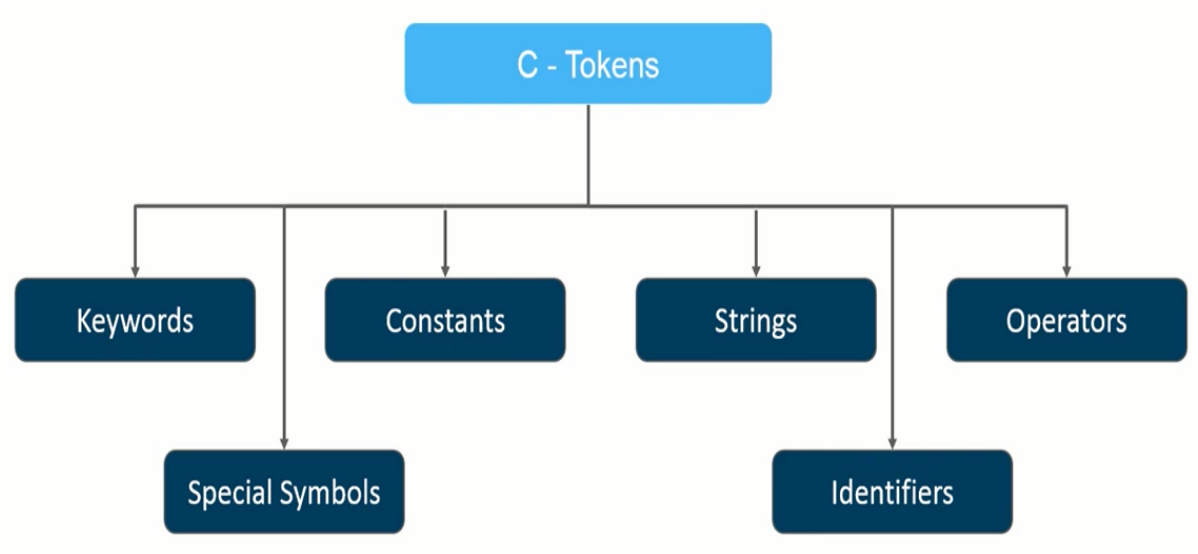



Figure 3 C-Tokens

Keywords: variables having specified meaning and are predefined in C library. Such as main, for, if, else etc. Keywords cannot be renamed or reprogramed. Figure 4 shows the 32 keywords in C language.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| auto | default | break | case | char | const | continue | do |
| double | else | enum | extern | float | for | goto | if |
| int | long | register | return | short | signed | sizeof | static |
| struct | switch | typedef | union | unsigned | void | volatile | while |

Figure 4 The 32 keywords in C language

Constants: sometimes refer to as literals are like variables, but unlike variables once they are declared their values cannot be changed. The syntax is:

const data_type variable_name;

Or

const data_type*variable_name;

Types of Constants in C language

1. Integer constants
2. Floating point constants
3. Character constants
4. String constants
5. Octal and Hexadecimal constants

Strings: these are collection of characters defined in form of an array and end with null character which describes the end of the string to the compiler. The syntax is:

char string_name[Length_of_the_string]

In strings we have alphabetical data meaning (A - Z) which is stored in form of arrays.

Identifiers: this are names declared in the program in order to name a value, variable function, array etc. The syntax is:

```
int x = 10;
```

In the above, x is the identifier and the value stored is 10. The keyword is int; this is the data type specified for the identifier x. meaning the value is an integer.

Rules for declaring an identifier

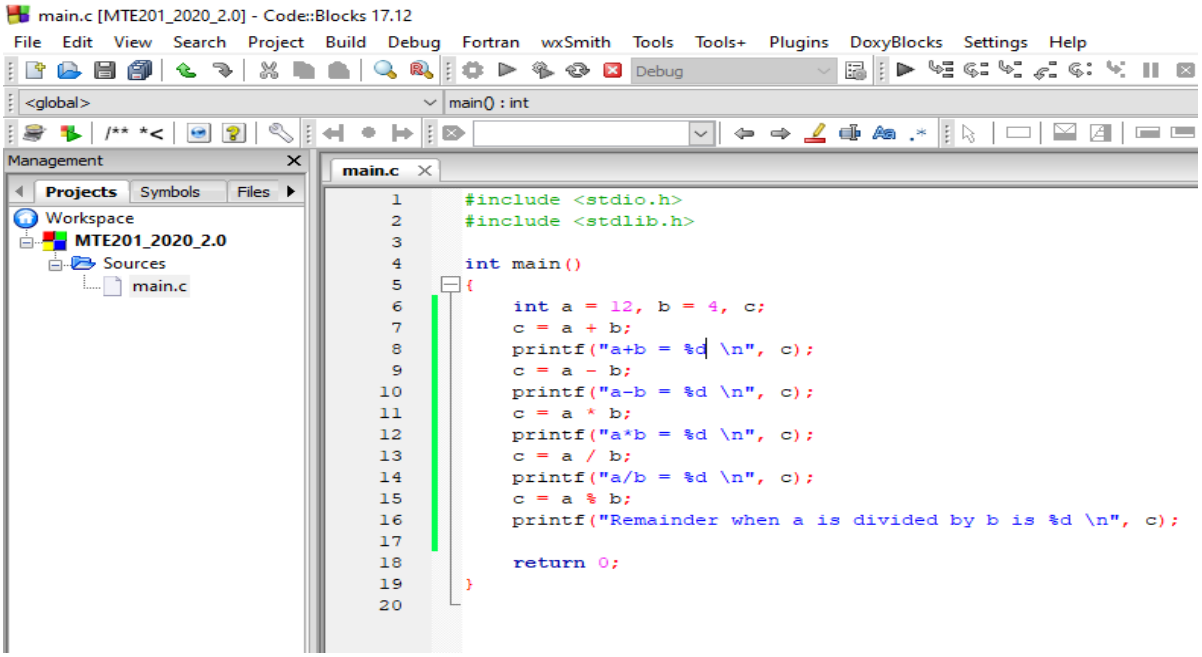
1. First character should be an alphabet or underscore.
2. Succeeding character can be digits or letters.
3. Special characters are not allowed except underscore.
4. Identifiers should not be keyword.

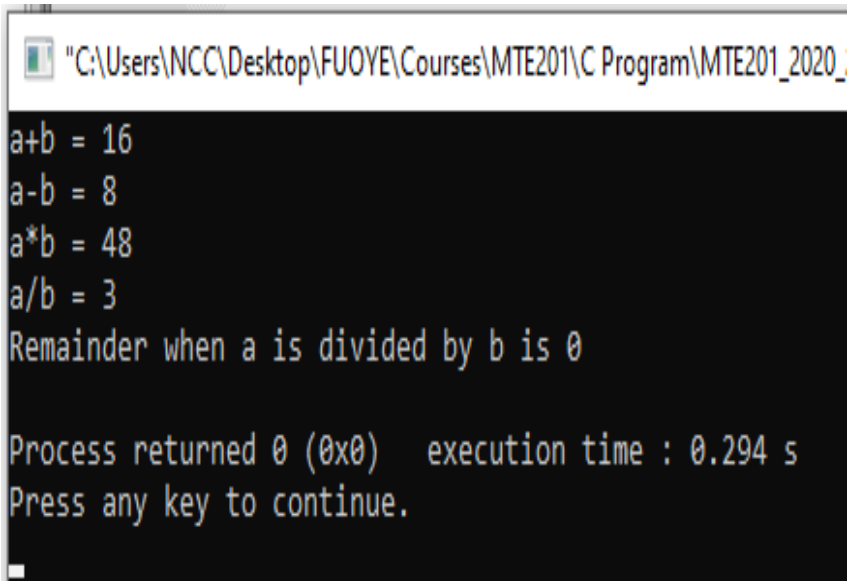
| Valid Identifiers names: | Invalid Identifiers names: |
|---|---|
| int a; | int 2; |
| int _ab; | int a b; |
| int a30; | int long; |

Special Symbols or Characters: this can be single character or sequence of characters having a special built-in meaning in language and typically cannot be used as identifiers. Such as: &, %, # etc. These special characters have meanings which are predefined in C library when designing the language, these is why they are used in particular segment of the code. For instance, the & is only used in printif and scanner statement, the % is used to specify the data type (integer data type %d, spring data type %S)

Operators: The following are the operators in C language

1. Arithmetic operators: these are used to perform mathematical operations. Such as addition, subtraction, multiplication and modulus.




2. Increment/decrement operators: These operators are used when loops are included in the program. Increment operators are use to increase the value of a variable by a specific number. Decrement operators are use to decrease the value of a variable by a specific number.

Example:
i++; // increment
i--; // decrement

3. Assignment operators: These operators are used to assign values to variable in C language.
   Example
   =, ==
   int b = 4;
4. Bitwise operators: These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
   Bit wise operators in C language are:
   & (bitwise AND)
   | (bitwise OR)
   ~ (bitwise NOT)
   ^ (XOR)
   << (left shift)
   >> (right shift)

5. Relational operators: These operators are used to test or define relation between two entities or variables.
   Example
   <, >, =, !=,

   If (a=<b);

| Operator | Meaning | Example |
|---|---|---|
| == | Equal to | `count == 10` |
| != | Not equal to | `flag != DONE` |
| < | Less than | `a < b` |
| <= | Less than or equal to | `low <= high` |
| > | Greater than | `pointer > end_of_list` |
| >= | Greater than or equal to | `j >= 0` |

6. Logical operators: These operators are used to perform logical operation on a given expressions. There are three logical operators in C language:
   Logic AND (&&)
   Logic OR (||)
   Logic NOT (!)

## 1.8. Datatype and Variable Used in C Programming Language

### 1.8.1. Datatype

There are four datatypes in C language:

1. Basic datatype

| Type | Constant Examples | printf chars |
|---|---|---|
| char | 'a', '\n' | %c |
| _Bool | 0, 1 | %i, %u |
| short int | — | %hi, %hx, %ho |
| unsigned short int | — | %hu, %hx, %ho |
| int | 12, -97, 0xFFE0, 0177 | %i, %x, %o |
| unsigned int | 12u, 100U, 0XFFu | %u, %x, %o |
| long int | 12L, -2001, 0xffffL | %li, %lx, %lo |
| unsigned long int | 12UL, 100ul, 0xffeeUL | %lu, %lx, %lo |
| long long int | 0xe5e5e5e5LL, 50011 | %lli, %llx, &llo |
| unsigned long long int | 12ull, 0xffeeULL | %llu, %llx, %llo |
| float | 12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1 | %f, %e, %g, %a |
| double | 12.34, 3.1e-5, 0x.1p3 | %f, %e, %g, %a |
| long double | 12.34l, 3.1e-5l | %Lf, $Le, %Lg |

2. Derived datatype
3. Enumeration datatype
4. Void datatype

Table 1 presents these datatype examples

| Type | Datatype |
|---|---|
| Basic Datatype | int, char, float, double |
| Derived Datatype | array, pointer, structure, union |
| Enumeration Datatype | enum |
| Void Datatype | void |

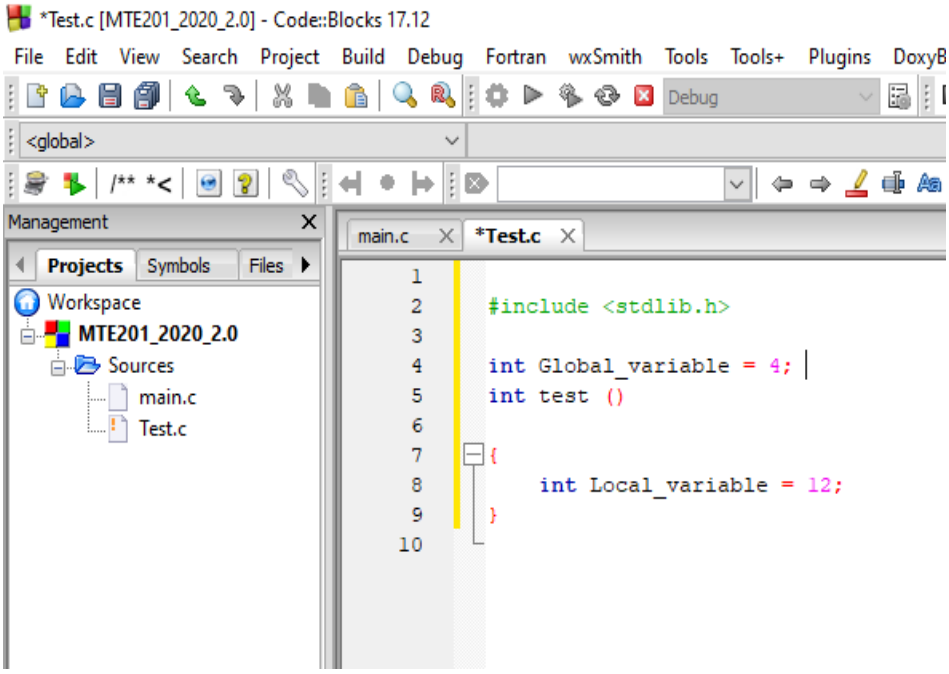
### 1.8.2. Variable

Variables are defined as the reserved memory space which stores value of a definite datatype. The value of the variable is not constant, so it can be changed. The types of variables in C language are:

1. Local variable: these variables are declared inside a code block or a function and has it scope limited to that particular block of code or function.


In the figure above, the function test is defined and a local variable with int datatype is declare within the function.

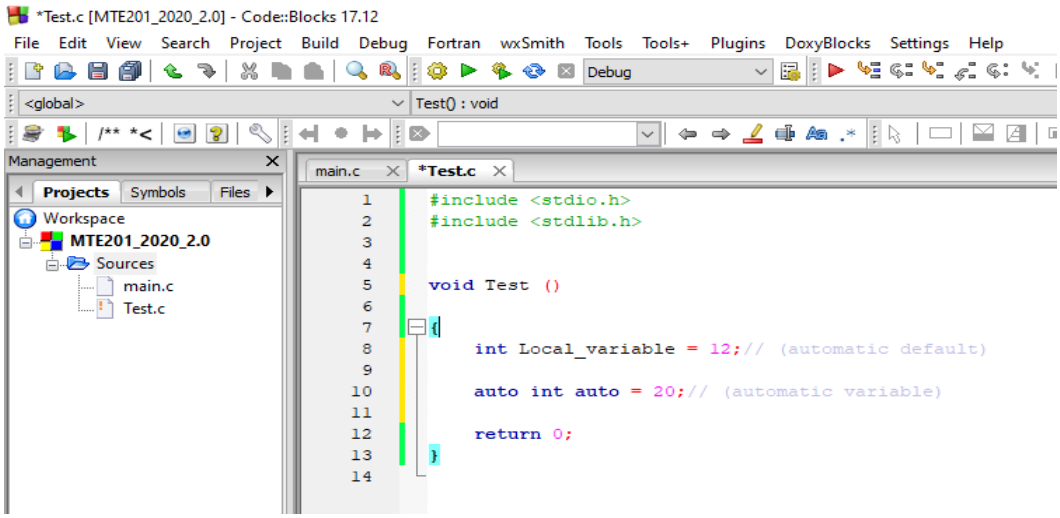
2. Global variable: these variables are declared outside a code block or a function and has it scope across the entire program and allow any function to change it value.




In the figure above, the function test is defined and a global variable with int datatype is declare outside the function.

3. Static variable:  any variable declared with the keyword static is known as static variable. These variables retain their declared value throughout the entire execution of the program and will not be changed between multiple function calls.
4. External variable: These variables are declared by using the keyword extern. A variable can be share between multiple C source file by using external variable.
   Example

   extern int extern = 10;//(External Variable)
5. Automatic variable: These variables can be declared by using the keyword auto. By default, all the variables define in C language are Automatic Variable.




Rules for declaring a Variable

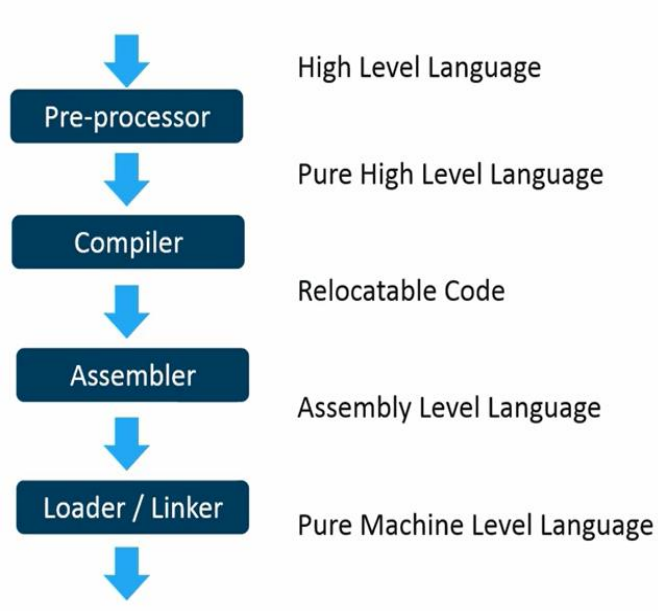
1. A variable can have alphabet, digit and underscore.
2. A variable name can start with alphabet and underscore only.
3. No spacing is allowed within the variable name.
4. A variable name must not be any reserve word or keyword.

| Valid Identifiers names: | Invalid Identifiers names: |
| --- | --- |
| int a; | int 2; |
| int _ab; | int a b; |
| int a30; | int long; |

## 1.9. PREPROCESSOR DIRECTIVES

These are lines included in a program that begin with the character #, which distinguish them from a typical source code text. They are invoked by the complier to process some programs before compilation. It is a macro processor used automatically by the C compiler to transform user program before actual compilation. For instance, stdio.h, means standard input/output; this preprocessor directive activates the input and output unit to perform a C programming operation.

The operating sequence of the preprocessor directive is shown below:




Types of Preprocessor Directives in C Language

1. #include: this used to paste code of given file into current file. It is used include system defined and user defined header files. There are two variants to use the #include directive:
   #include <filename>
   #include "filename"
2. #define: this is used to define constant or macro substitution. It can use any basic datatype. Syntax is #define token value.
3. #undef: is used to undefine the constant or macro defined by #define. The syntax is #undef token
4. #ifdef
5. #ifndef
6. #else
7. #error

8. #pragma

# 2.0. Control statements and Loops

## 2.1. Control statements

These statements enable user to specify the flow of program control. They specify the order in which the instruction in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or jump from one section of the code to another.

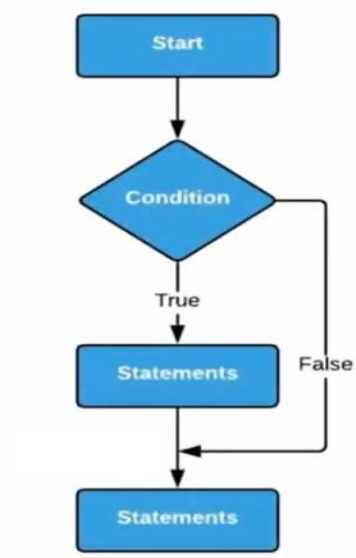
The variants of the control statements in C language are:

1. If statement
2. If-else statement
3. If-else ladder
4. Nested if
5. Switch
6. Ternary
7. Break

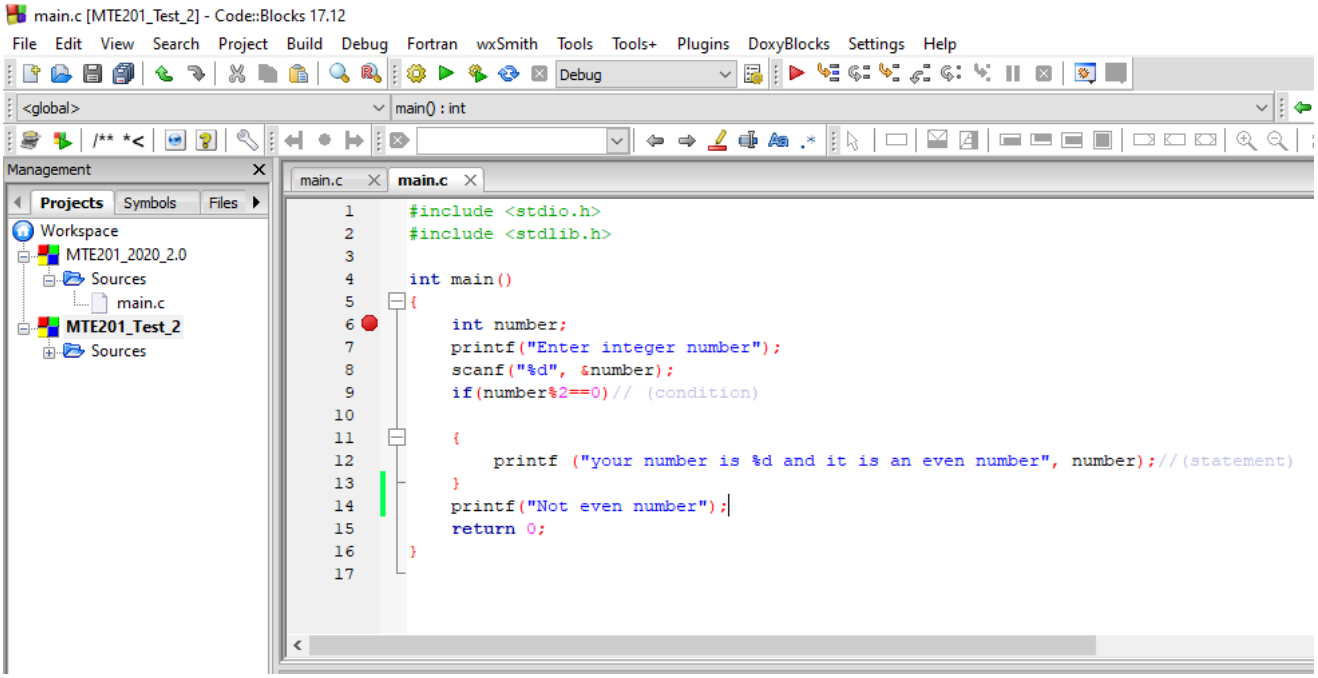
### 2.1.1. If control statement

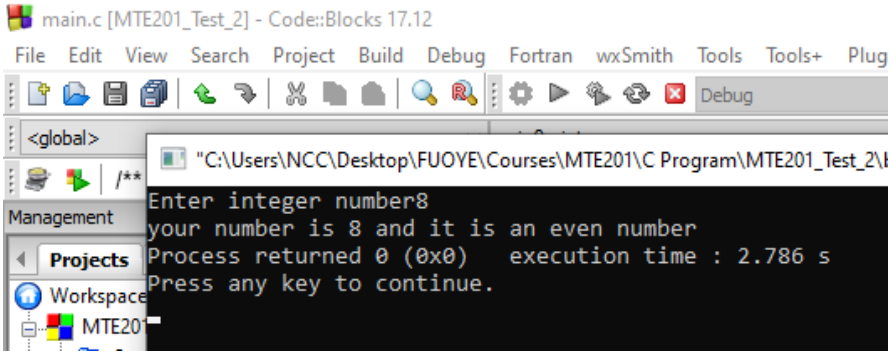
The if statement in C language is defined as a programming conditional statement that, if proved true, it performs an operation or displays information inside the statement block.

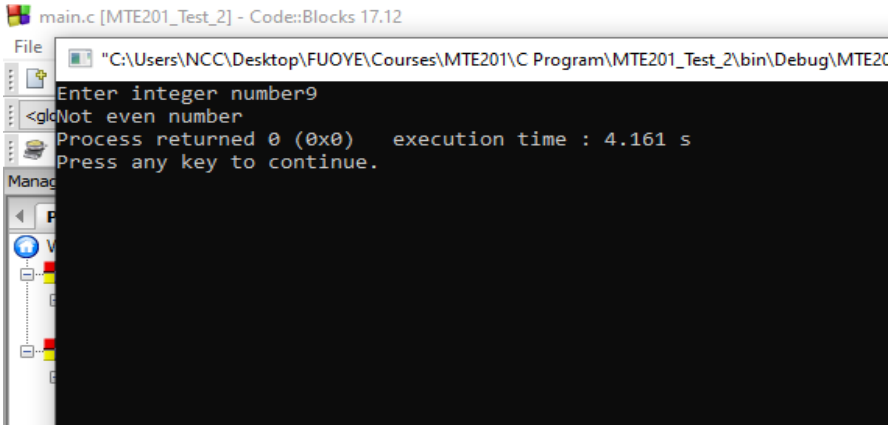
This can be explained using the flow chart




For better understanding let write the code:


main.c [MTE201_Test_2] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
main() : int
Management
Projects Symbols Files
Workspace
MTE201_2020_2.0
Sources
main.c
MTE201_Test_2
Sources
main.c
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int number;
printf("Enter integer number");
scanf("%d", &number);
if(number%2==0)// (condition)
{
printf ("your number is %d and it is an even number", number);//(statement)
}
printf("Not even number");
return 0;
}

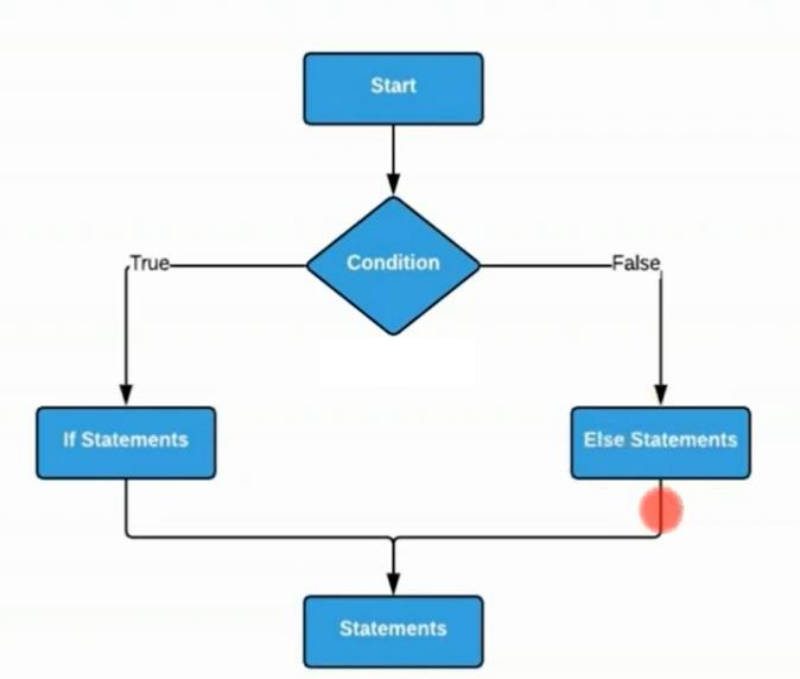

main.c [MTE201_Test_2] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Debug
<global>
"C:\Users\NCC\Desktop\FUOYE\Courses\MTE201\C Program\MTE201_Test_2\
Enter integer number8
your number is 8 and it is an even number
Process returned 0 (0x0) execution time : 2.786 s
Press any key to continue.
Management
Projects
Workspace


main.c [MTE201_Test_2] - Code::Blocks 17.12
File
"C:\Users\NCC\Desktop\FUOYE\Courses\MTE201\C Program\MTE201_Test_2\bin\Debug\MTE2
Enter integer number9
Not even number
Process returned 0 (0x0) execution time : 4.161 s
Press any key to continue.

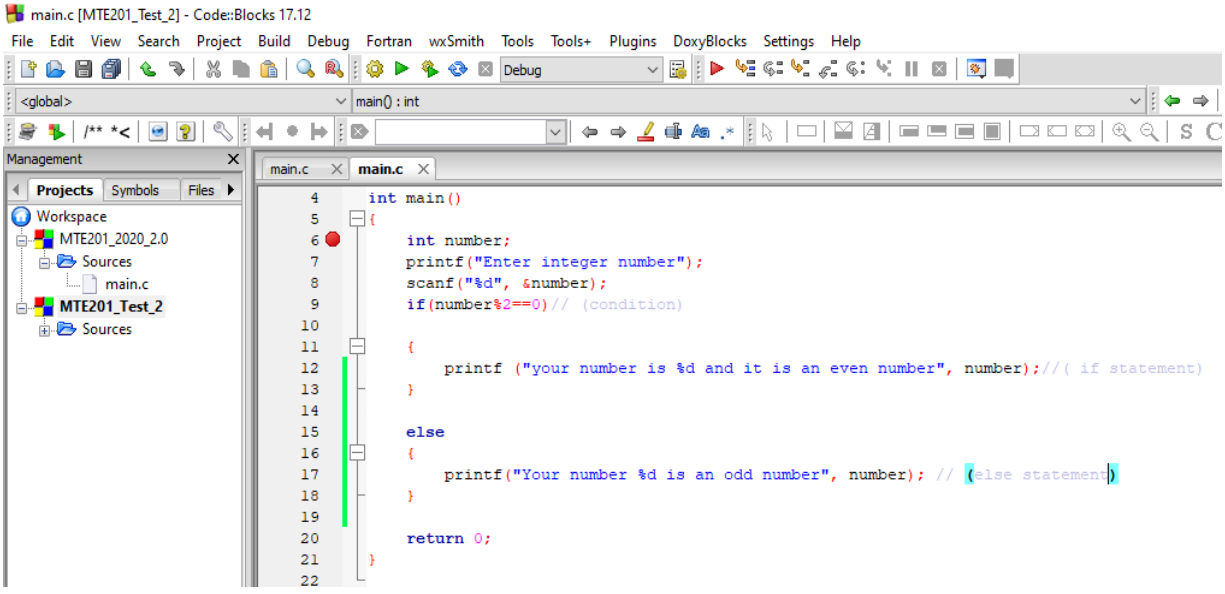
### 2.1.2. If-else control statement

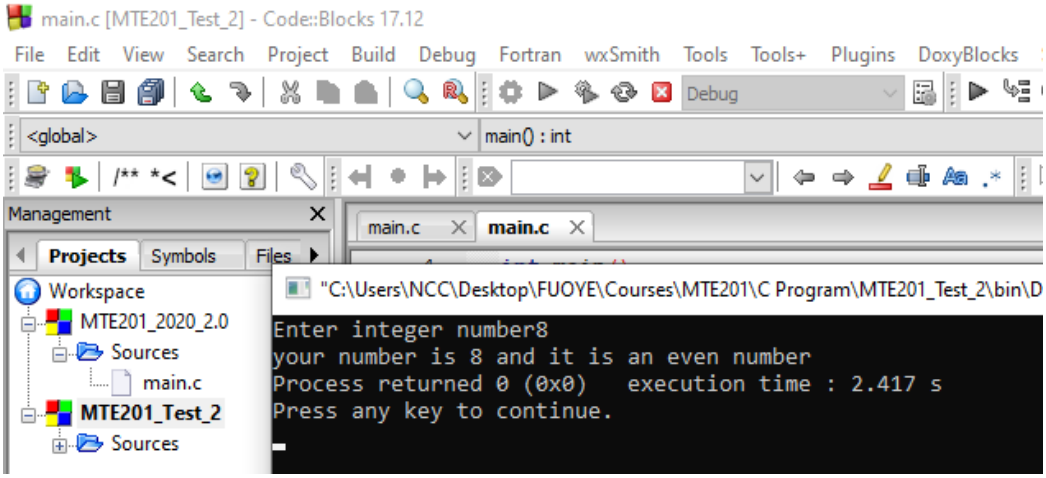
This control statement defines a programming conditional statement that has two statement blocks over a condition. If proved true, then the **if** block is executed and if false, then the **else** block is executed.

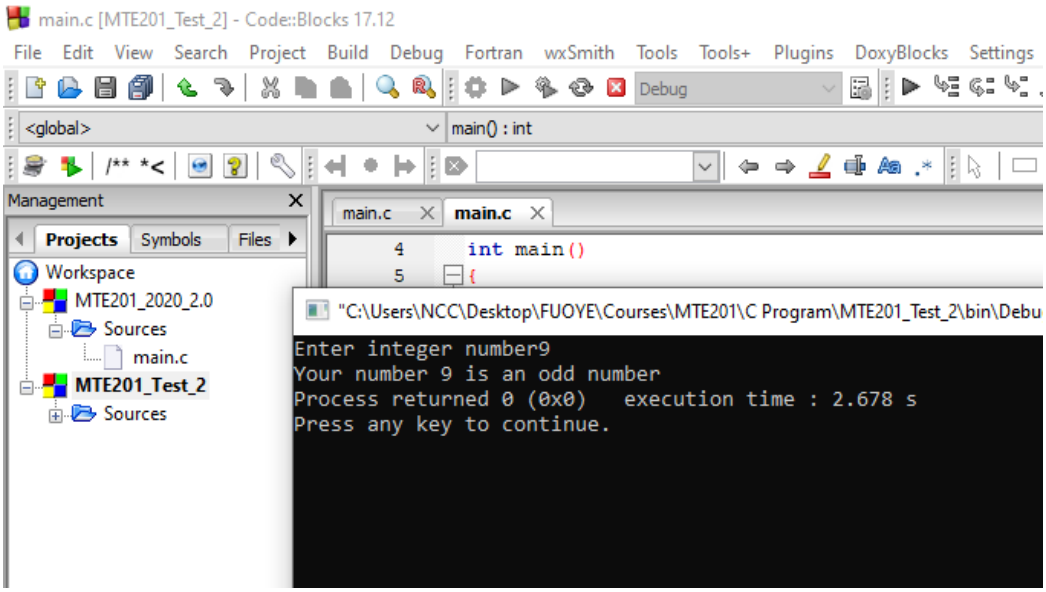
This can be explained using the flow chart




For better understanding let write the code:



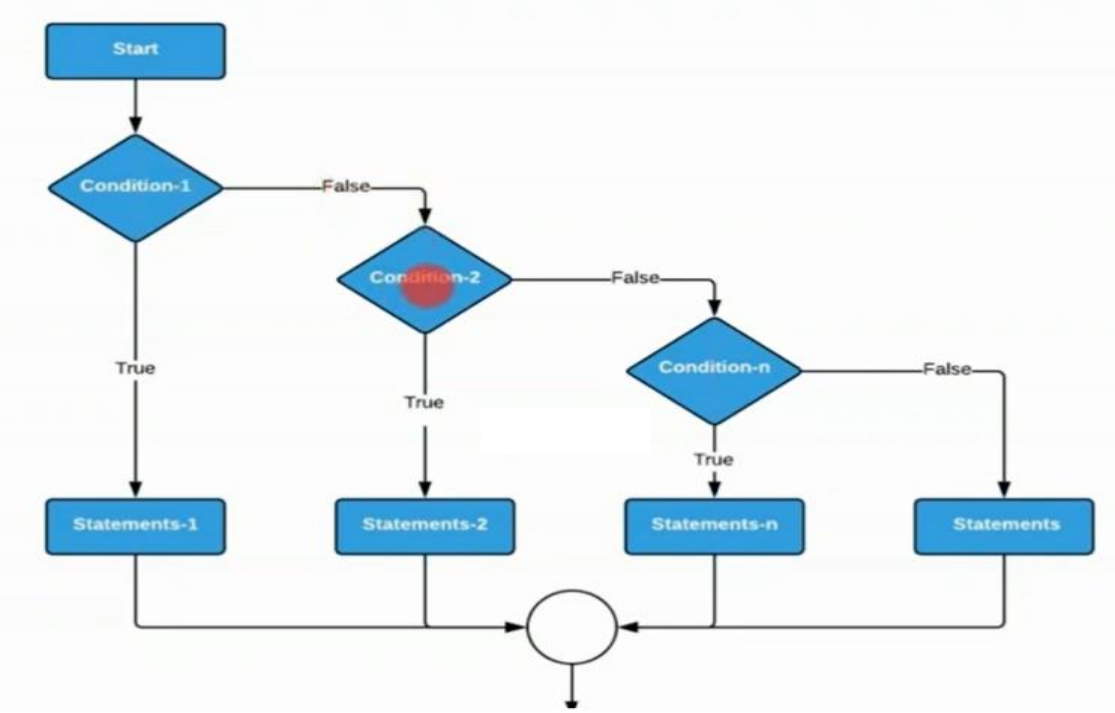


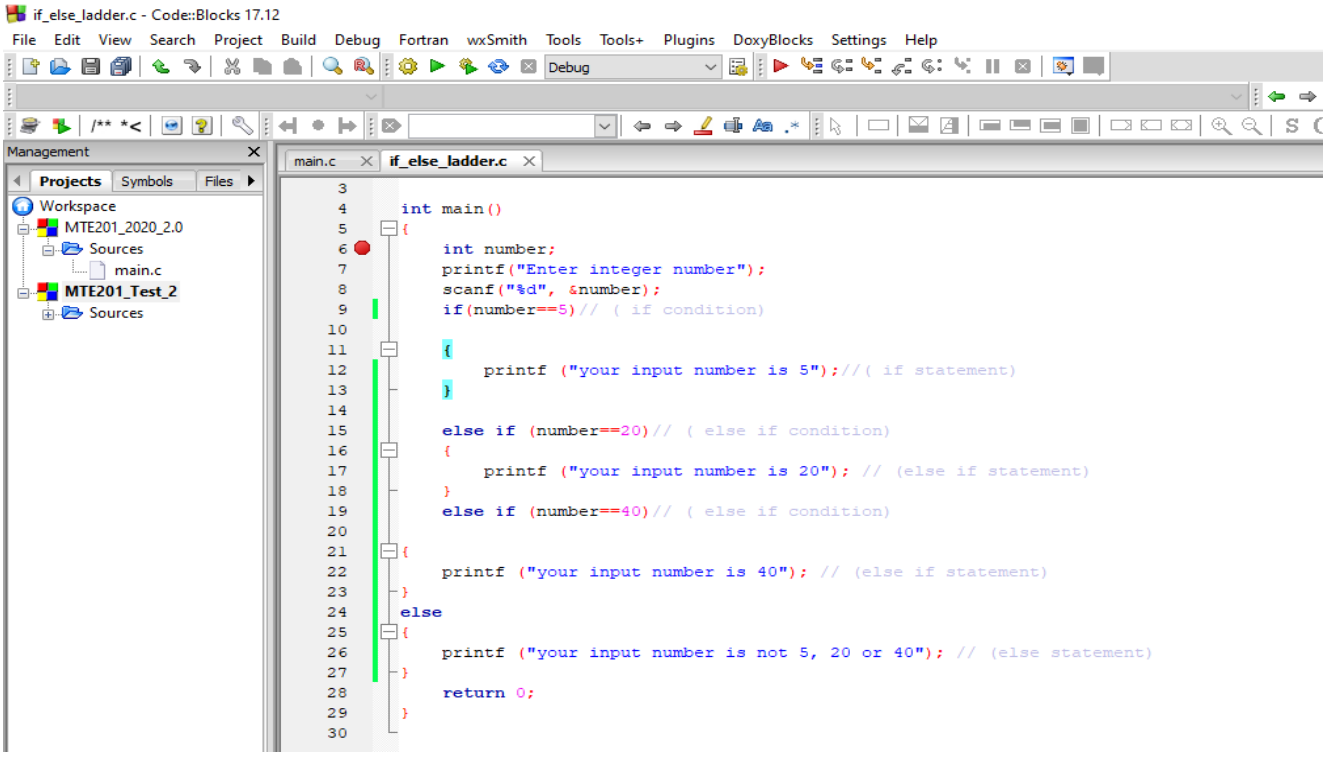

### 2.1.3. If-else ladder Control Statement

This control statement defines a programming conditional statement that has multiple else-if statement blocks. If any of the condition is true, then the control will exit the else-if ladder and execute the next set of statements.

This can be explained using the flow chart

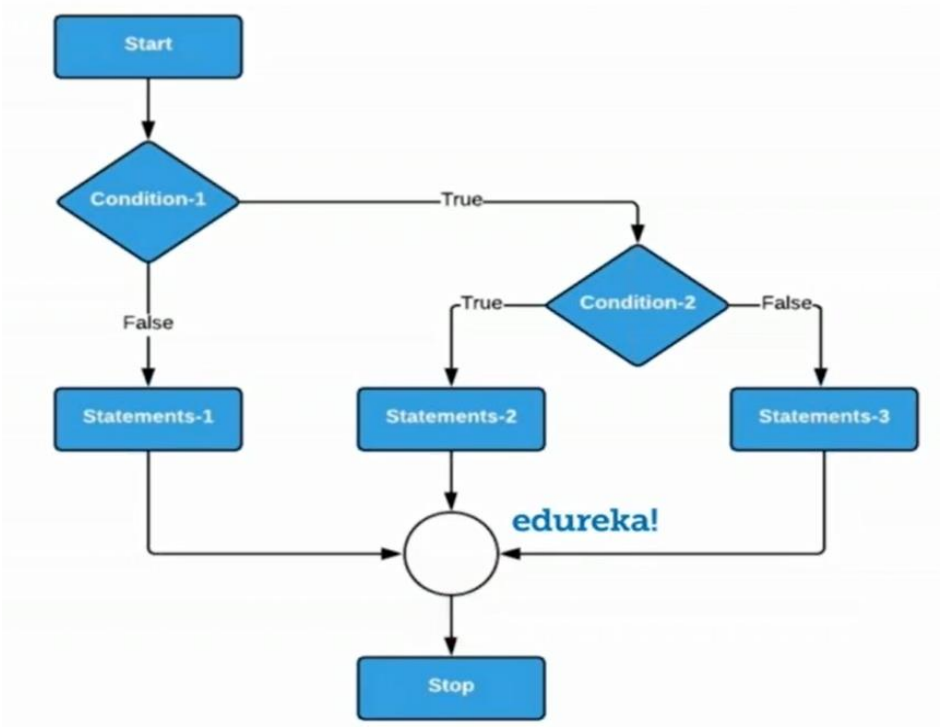



For better understanding let write the code:

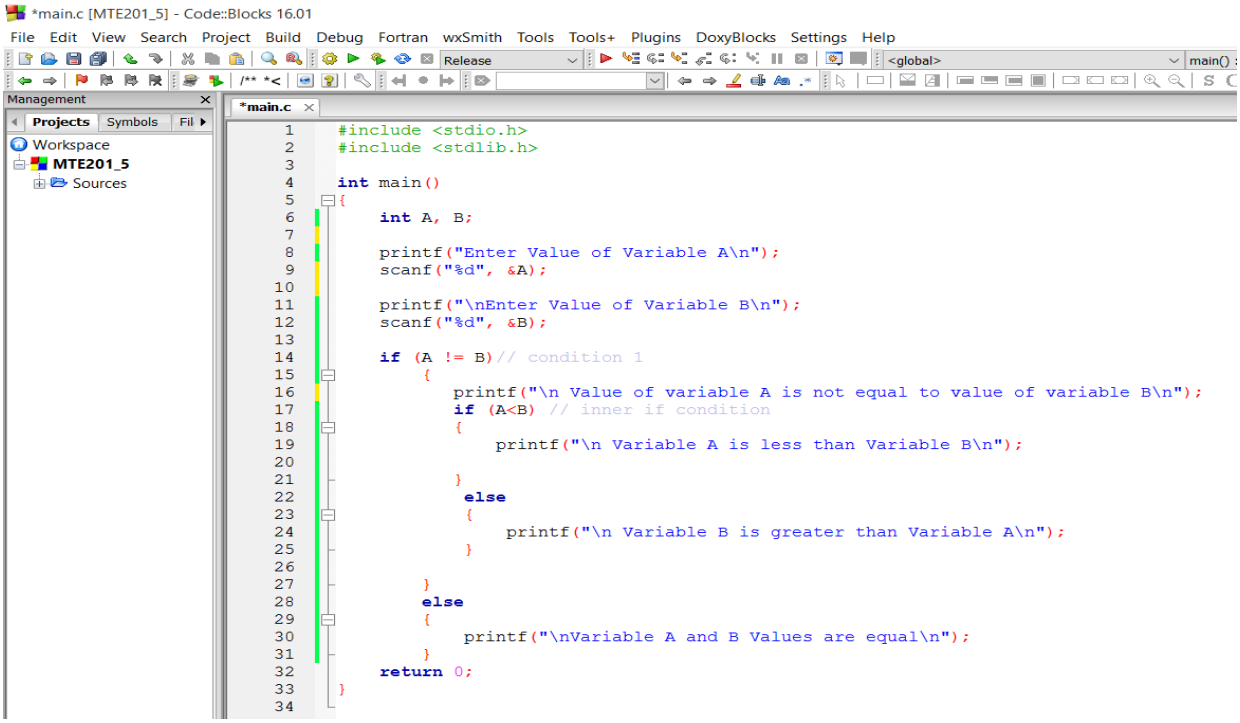


### 2.1.4. Nested If control statement

This is a programming conditional statement that consist of another if statement within the previous if statement.

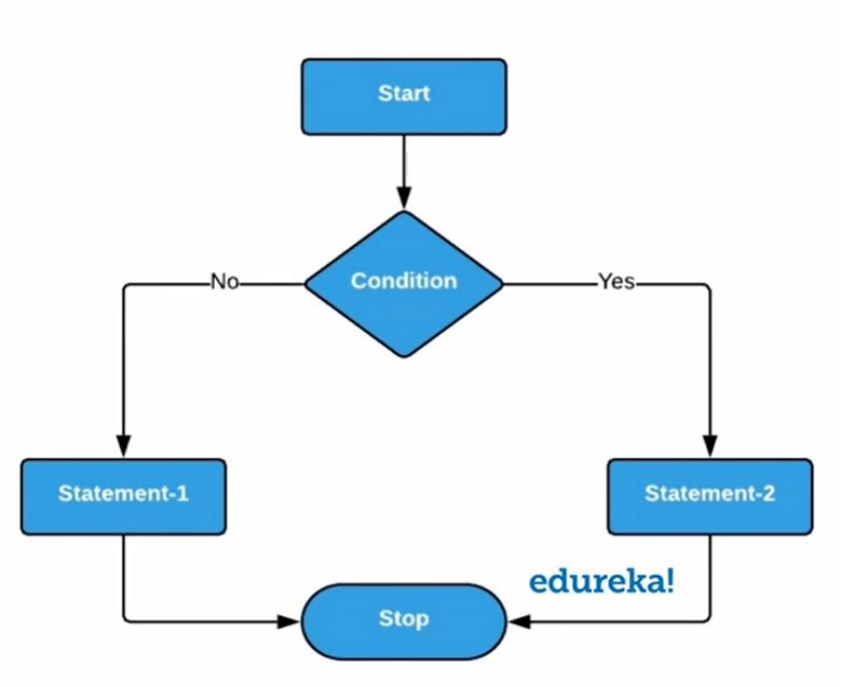



Nested If control statement flowchart

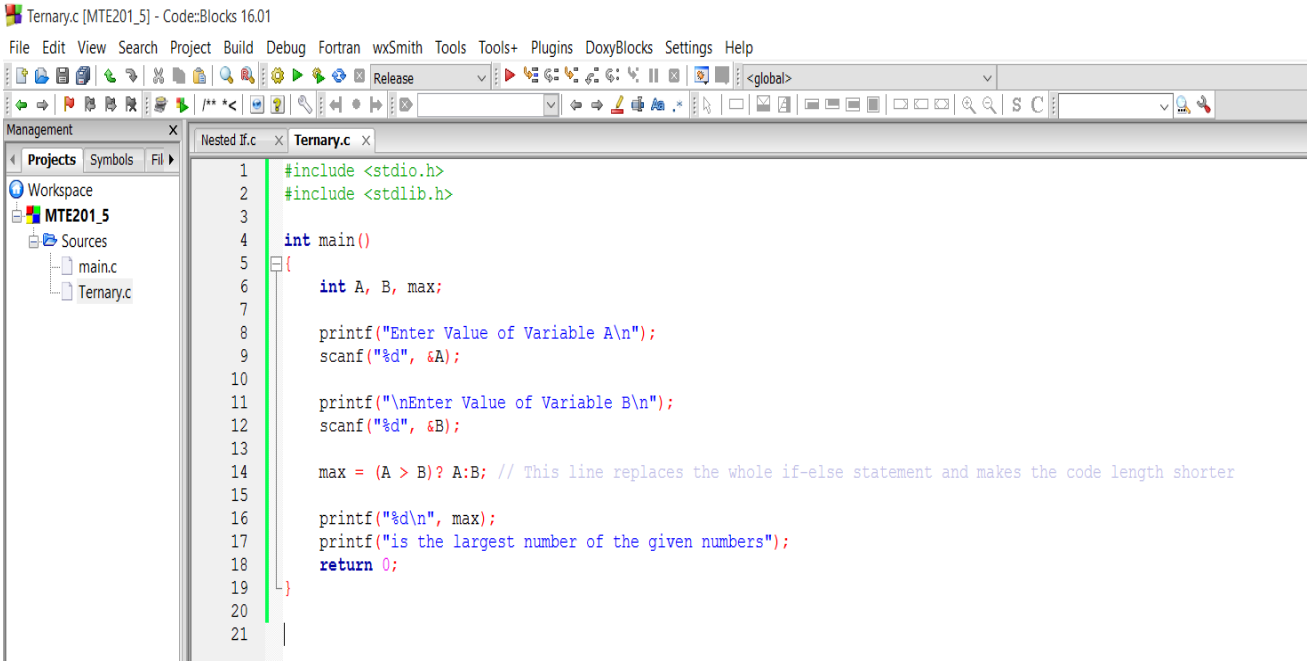


### 2.1.5. Ternary Control Statement

This is a programming conditional statement that is similar to if-else statement but shorter in code length. In this case the control checks the conditions and executes either of the two statements.




Ternary Control Statement Flowchart



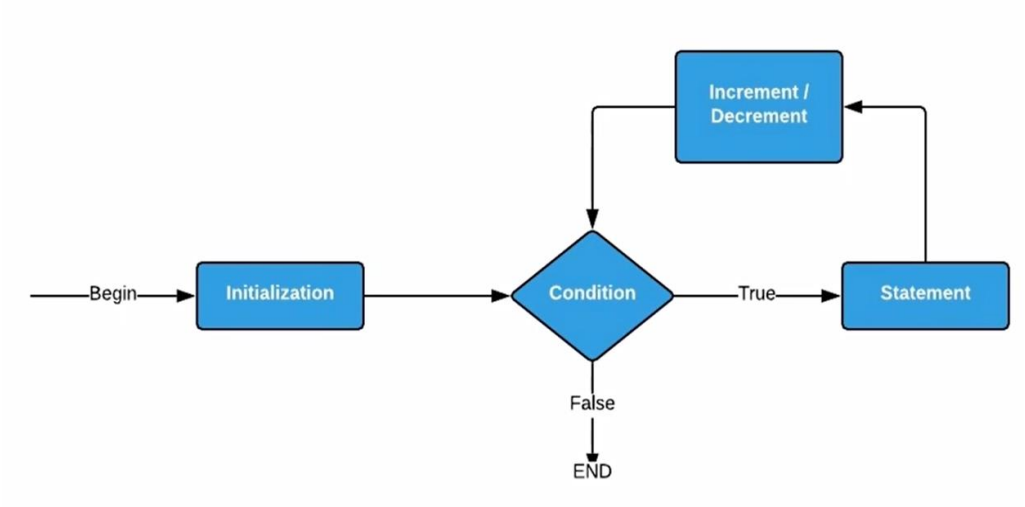
## 2.2. Loops

These are control statements used in C language to perform looping operations while a give condition is true. Control exit the loop once the given condition is false. The three types of loops in C language are:

1. For loop
2. While loop
3. Do while loop

### 2.2.1. For Loop

This is a precise loop with the syntax: initialization, condition and increment or decrement operator. The flowchart is presented below:




For Loop flowchart

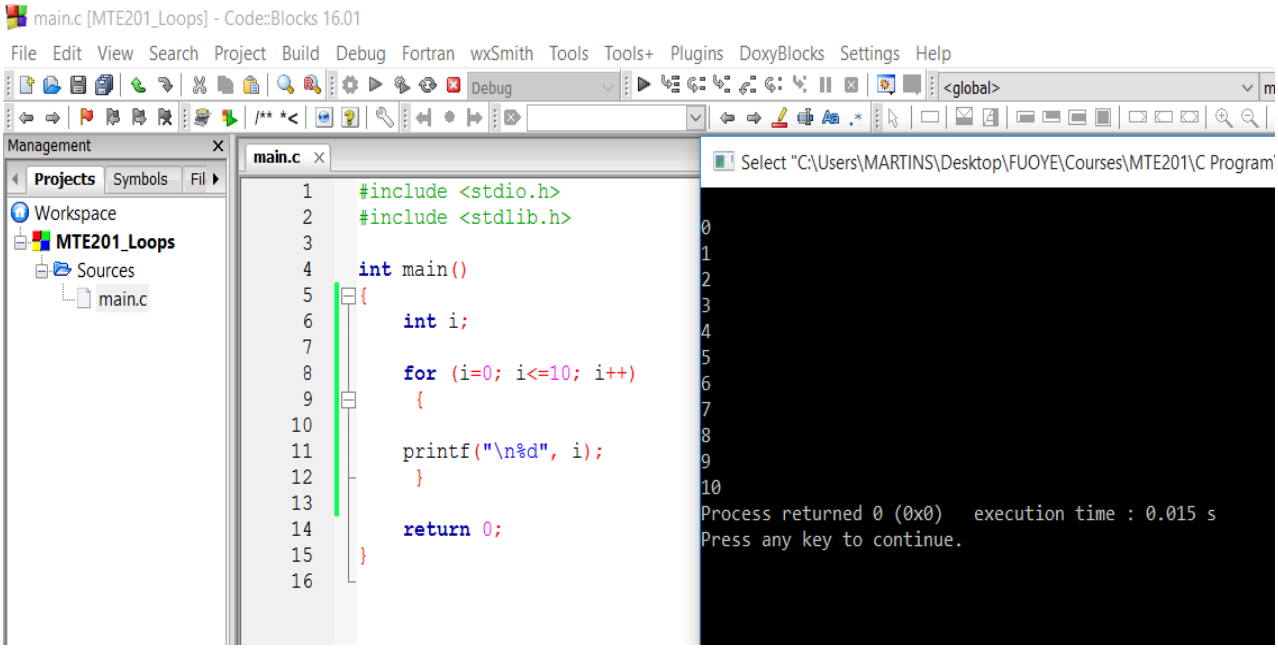
For instance, the code:

```
for (i=0; i<=10; i++)
```

i=0 is the initialization value for i

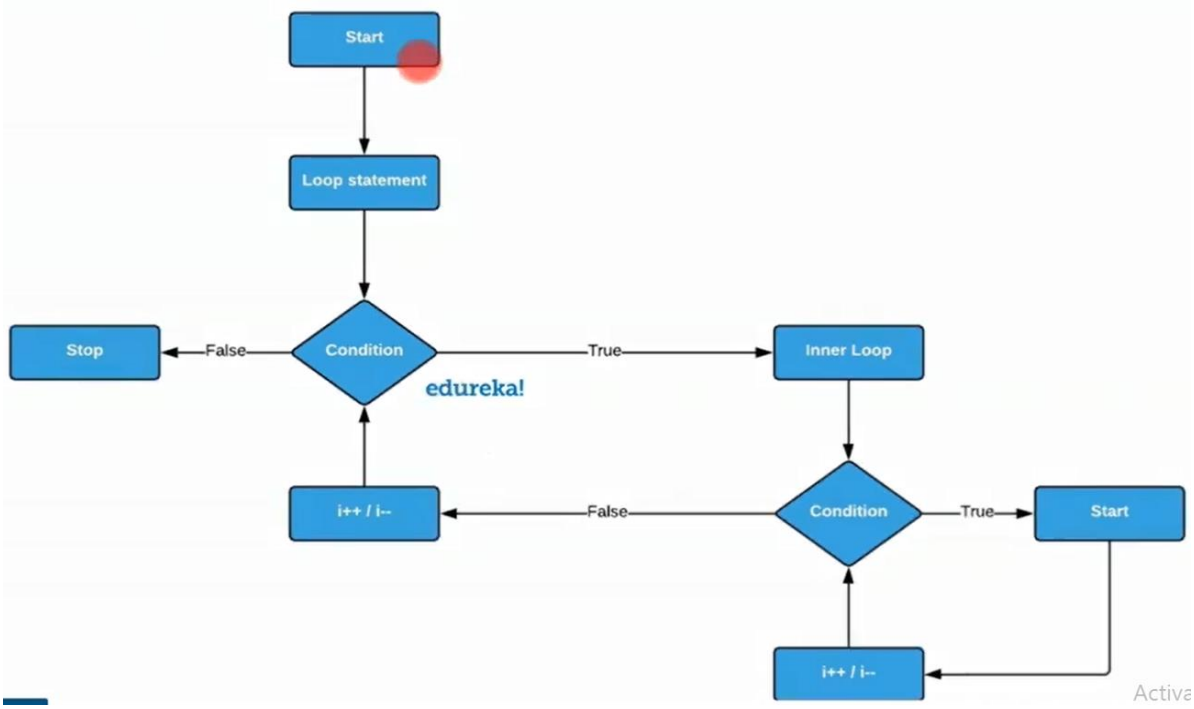
i<=10 is the condition

i++ is the increment operator

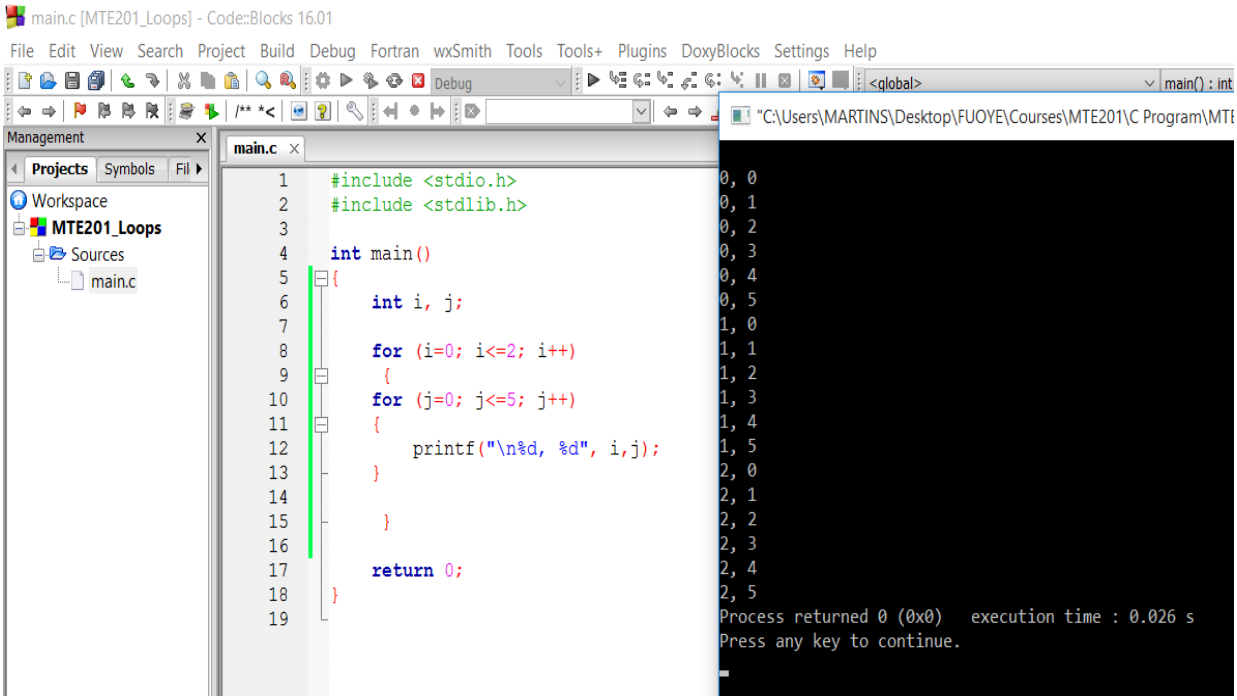



## 2.2.2. Nested For Loop

This when a for loop is within an existing for loop. The flowchart below depicts this situation:

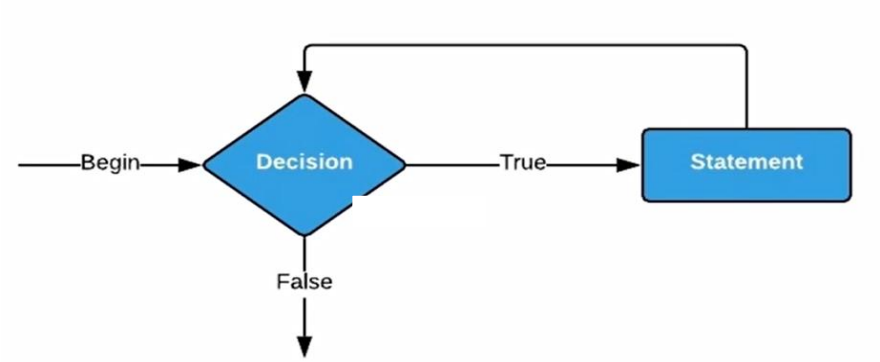


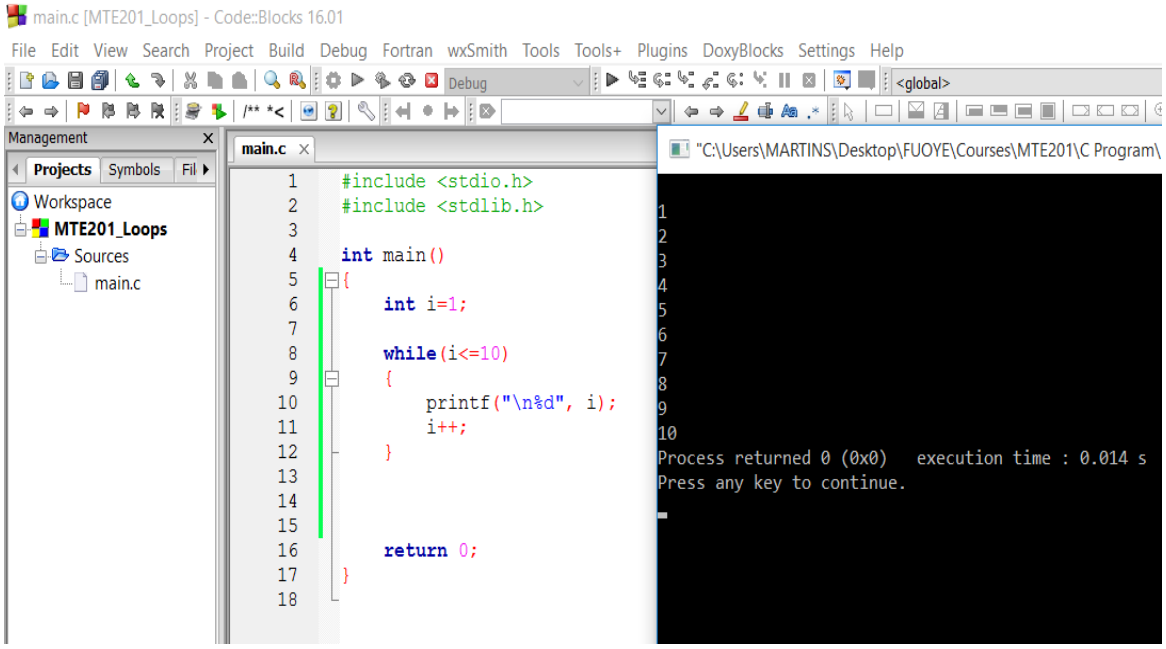

Nested For Loop flowchart




## 2.2.3. While Loop

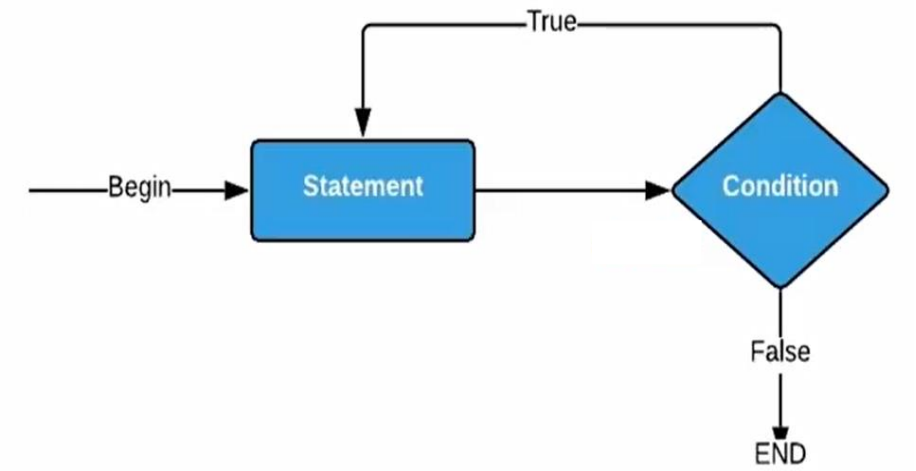
This loop execute itself repeatedly until a given Boolean expression or condition is true.



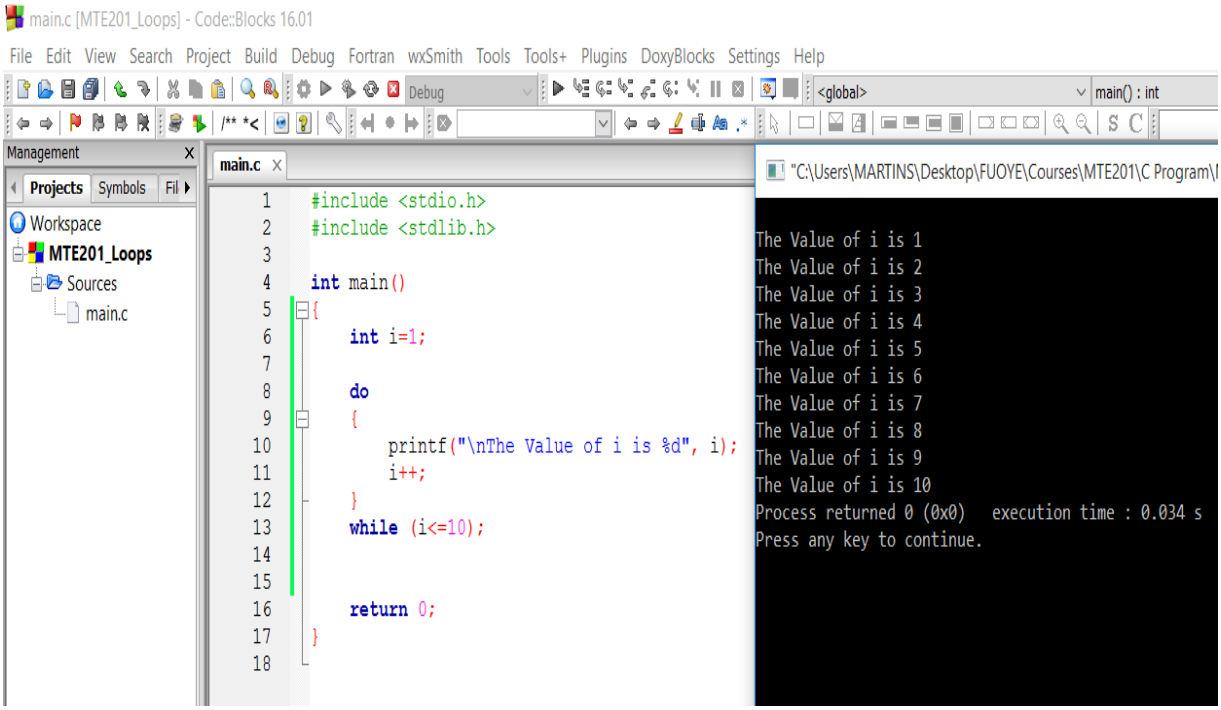



### 2.2.4. Do While Loop

The difference between the Do While loop and While loop is that the condition is stated at the end of the loop.




Do While Loop flowchart


main.c [MTE201_Loops] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
main() : int
Management
Projects
Symbols
Workspace
MTE201_Loops
Sources
main.c
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int i=1;
do
{
printf("\nThe Value of i is %d", i);
i++;
}
while (i<=10);
return 0;
}
"C:\Users\MARTINS\Desktop\FUOYE\Courses\MTE201\C Program\
The Value of i is 1
The Value of i is 2
The Value of i is 3
The Value of i is 4
The Value of i is 5
The Value of i is 6
The Value of i is 7
The Value of i is 8
The Value of i is 9
The Value of i is 10
Process returned 0 (0x0) execution time : 0.034 s
Press any key to continue.

## 3.0. Pointers

These are variables which stores the address of the value of another variable. This variable can be of any type e.g. int, char, array etc. the size of the pointer depends on the architecture. The pointer in C language can be declared using *(asterisk symbol)

The syntax is:

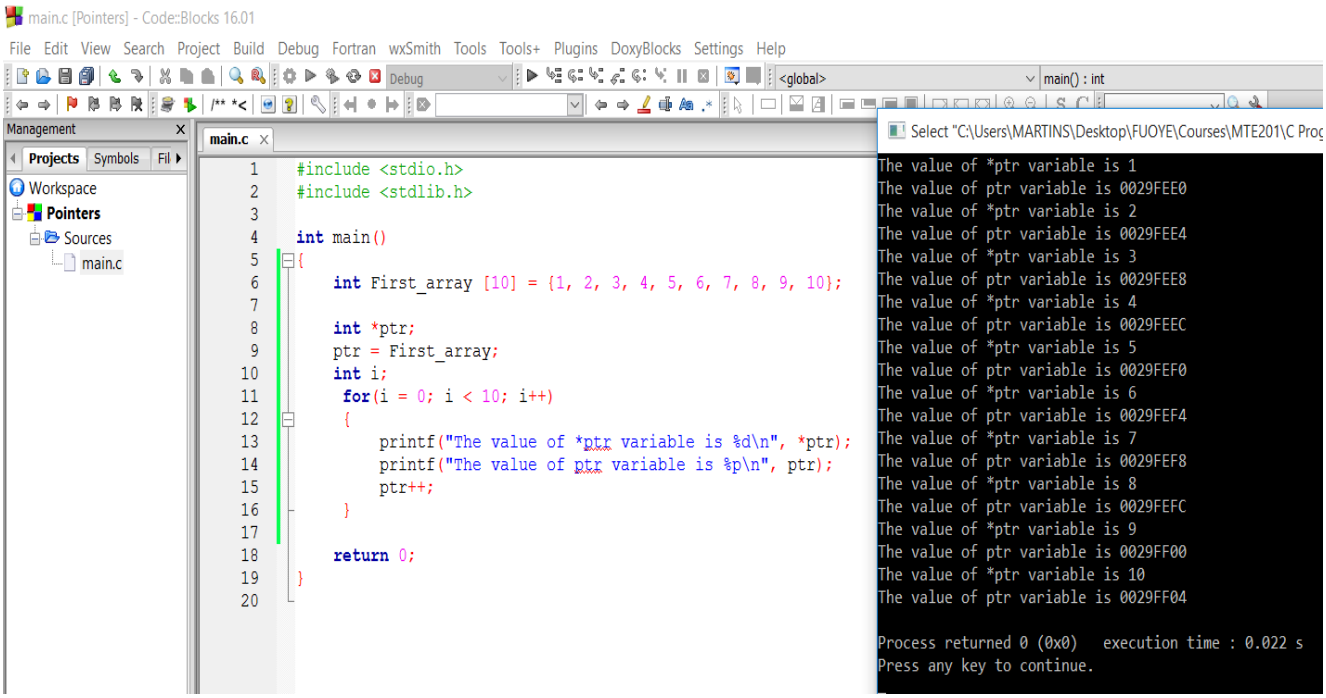
float y = 2.1;

float*u = &y;

In the above, the value stored in y is 2.1, using the pointer function we can store the address of the value stored in y directly.

The merits of pointers are:

1. User can return multiple values from a function
2. User can access any memory location in the computer's memory
3. User can dynamically allocate memories using the malloc() and callc() function using pointers
4. Pointers in C language are widely used in arrays, functions and structures.
5. It reduces the code and improves performance



### 3.1. Escape Sequence

These can be defined as combination of backward slash (\) and a letter or digit. These sequences are non-printable and are used to communicate with display devices or printer by sending non-graphical control characters to specify actions like new line and tab space.

Listed in the below are the escape sequence in C language.

| Escape Sequence | Functionality |
|---|---|
| \a | Alarm |
| \b | Backspace |
| \f | Form Feed |
| \n | New Line |
| \r | Carriage Return |
| \t | Tab Space |
| \v | Vertical Tab |

| Escape Sequence | Functionality |
|---|---|
| \\ | Back Slash |
| \' | Single Quote |
| \" | Double Quote |
| \? | Question Mark |
| \nnn | Octal Number |
| \xhh | Hexadecimal Number |
| \0 | Null |

### 3.2. Functions

These can be defined as a subdivision program of a main program enclosed within flower brackets. Functions can be called by the main program to implement its functionality. This procedure provides code reusability and modularity. The two functions in C language are:

1. Library function
2. User-defined functions

The advantages of functions are:

1. User can avoid writing the same code repeatedly in a program

2. Functions can be called any number if times in a program
3. Program tracking is easier when it is divided into multiple functions
4. Reusability is the main achievement of C functions.

Rules for using functions in C language

1. Function declaration: a function is required to be declared as *Global* and the *name* of the function, *parameters* of the function and the *return type* of the function are required to the clearly specified.
2. Function call: while calling the function from anywhere in the program, car should be taken that the *data type* in the argument list and *number of elements* in the argument list are matching
3. Function definition: after the function is declared, it is important that the function includes *parameters declared*, *code segment* and the *return value*.

How to use functions in C language?

1. Functions without arguments and without return values
2. Functions without arguments and with return values
3. Functions with arguments and without return values
4. Functions with arguments and with return values

Calling a function can be by value:

main.c

```c
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);
    printf("After function call x=%d \n", x);
return 0;
}
```

input

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Calling function by reference:

main.c

```c
#include <stdio.h>
void swap(int , int);
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b);
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b);
}
```

input

```
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20


...Program finished with exit code 0
Press ENTER to exit console.
```

# Introduction to Analysis Software (OCTAVE)

## 3.0. Getting Started into OCTAVE

OCTAVE is a *mathematical and graphical software package*; it has numerical, graphical, and programming capabilities. It has built-in functions to do many operations, and there are toolboxes that can be added to augment these functions (e.g., for signal processing). There are versions available for different hardware platforms, and there are both *professional and student editions.*

OCTAVE is started just like any other Windows program. When the OCTAVE software is started, a window is opened (See Figure 1.1). As shown in the figure, the screen is divided into three main elements. These are

1. File listing in the current directory
2. Command History Window
3. Command Window (the main part is the Command Window)

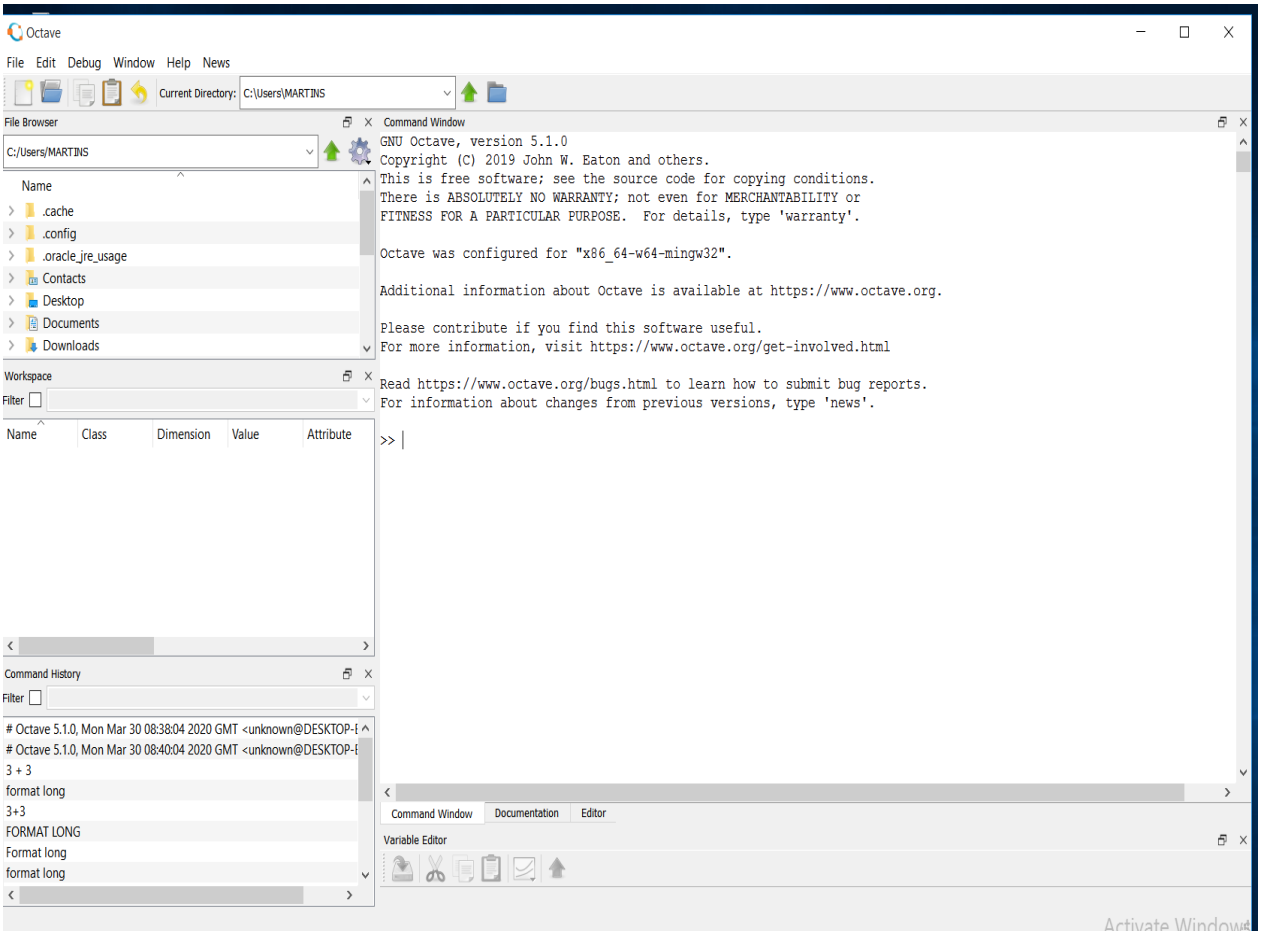



Fig. 1.1: The OCTAVE desktop

In the Command Window, you should see:

>>

The >> is called the ***prompt***. In the Student Edition, the prompt appears as:

EDU>>

In the Command Window, OCTAVE can be used interactively. At the prompt, any OCTAVE command or expression can be entered, and OCTAVE will immediately respond with the result.

It is also possible to write programs in OCTAVE, which are contained in **script files** or **M-files**.

The standard mix of menus appears on the top of the OCTAVE desktop that allows you to do things like file management and debugging of files you create. Also a drop-down list on the upper right side of the desktop allows you to select a directory to work in.

### 3.1. Performing Basic Arithmetic In The Command Window

The Command Window is on the right-hand side of the OCTAVE desktop. Commands are entered at the prompt with looks like two successive "greater than" signs:

>>

For instance, If you want to find the value of a numerical expression, simply type it in. Let's say we want to know the value of 433.12 multiplied by 15.7. We type 433.12 * 15.7 at the OCTAVE prompt and hit the enter key. The result looks like this:

```
>> 433.12*15.7
ans =
   6.8000e+003
```

OCTAVE prints out the answer to our query conveniently named **ans**. This is a *variable or symbolic name* that can be used to represent the value later. Chances are we will wish to use our own variable names. So for example, we might want to call a variable $x$. Suppose we want to set it equal to five multiplied by six. To do this, we type the input as

```
>> x=5*6
x =
   30
```

Once a variable has been entered into the system, we can refer to it later. Suppose that we want to compute a new quantity that we'll call $y$, which is equal to $x$ multiplied by 3.56. Then we type

```
>> y = x * 3.56
y =
   106.8000
```

Note: OCTAVE does not require you to include spaces in your input. But to enhance the readability and professional appearance of our output it is advised to include spaces in your input.

#### 3.1.1. Operator Precedence Rules

Here are some of the common operators that can be used with numeric expressions:

| | |
|---|---|
| + | addition |
| – | negation, subtraction |
| * | multiplication |
| / | division (divided by e.g. 10/5 is 2) |
| \ | division (divided into e.g. 5\10 is 2) |
| ^ | exponentiation (e.g., 5^2 is 25) |

Some operators have precedence over others. The precedence followed in mathematical operations by OCTAVE is the same used in standard mathematics, but with the following caution for left and right division. That is, exponentiation takes precedence over multiplication and division, which fall on equal footing. Right division takes precedence over left division.

Finally, addition and subtraction have the lowest precedence in OCTAVE. To override precedence, enclose expression in parentheses.

```
>> 4 + 5 * 3
ans =
   19
>> (4 + 5) * 3
ans =
    27
```

Within a given precedence level, the expressions are evaluated from left to right (this is called the ***associativity***).

**Nested parentheses** are parentheses inside of others; the expression in the inner **parentheses** is evaluated first. For example, in the expression 5 – (6 *(4 + 2)), first the addition is performed, then the multiplication, and finally the subtraction to result in -31. Parentheses can also be used simply to make an expression clearer. For example, in the expression ((4 + (3 * 5)) –1) the parentheses are not necessary, but are used to show the order in which the expression will be evaluated.

**Example 1.1**

1. Use OCTAVE to evaluate

$$5\left(\frac{3}{4}\right)+\frac{9}{5}$$

The command required to find the value of the first expression is:

```
>> 5*(3/4) + 9/5
ans =
   5.5500
```

2. Use OCTAVE to evaluate

$$4^3\left[\frac{3}{4}+\frac{9}{(2)3}\right]$$

The command required to find the value of the first expression is:

```
>> r = 4^3
r =
     64
>> s = 3/4 + 9/(2*3)
s =
     2.2500
>> t=r*s
t =
     144
```

**Practice 1.1**

Use OCTAVE to evaluate

```
4 ^ 2 - 1
4 ^ (2 - 1)
2\3
4 * 2 - 9/3
5 - - 3
```

### 3.1.2. Variables and Assignment Statements

In order to store a value in a OCTAVE session, or in a program, a ***variable*** is used. One easy way to create a ***variable*** is to use an ***assignment statement***. The format of an assignment statement is:

```
variablename = expression
```

The variable is always on the left, followed by the ***assignment operator***, **=** (unlike in mathematics, the single equal sign does not mean equality), followed by an expression. The expression is evaluated and then that value is stored in the ***variable***. For example, this is the way it would appear in the Command Window:

```
>> mynum = 6
mynum =
   6
```

Since the equal sign is the assignment operator, and does not mean equality, the statement should be read as "mynum gets the value of 6" (not "mynum equals 6").

Note that the variable name must always be on the left, and the expression on the right. An error will occur if these are reversed.

```
>> 6 = mynum
??? 6 = mynum
      |
Error: The expression to the left of the equals sign is not
   a valid target for an assignment.
```

Or

```
x + 6 = 90
```

OCTAVE will not make $X$ the subject of the formula but will return an error statement.

```
??? x+6=90
Error: The expression to the left of the equals sign is not
a valid target for an assignment.
```

The correct way is to type:

```
x = 90 - 6
```

To change a variable, another assignment statement can be used that assigns the value of a different expression to it. Consider, for example, the following sequence of statements:

```
>> mynum = 3
mynum =
        3
>> mynum = 4 + 2
mynum =
        6
```

```
>> mynum = mynum + 1
mynum =
         7
```

It obvious, that OCTAVE stores the most recent information about the particular ***variable*** name.

Secondly, to use a variable on the right-hand side of the assignment operator, we must assign a value to it previously. So while the following command sequence will generate an error:

```
>> x = 2
x =
      2
>> t = x + a
??? Undefined function or variable 'a'.
```

The correct way is:

```
>> x = 2
x =
      2
>> a = 3.5
a =
      3.5000
>> t = x + a
t =
      5.5000
```

In many instances, it is not desirable to have OCTAVE spit out the result of an assignment. To suppress OCTAVE output for an expression, simply add a semicolon (;) after the expression.

Consider the cases below:

```
>> x = 3
x =
      3
>>
```

```
>> x = 3;
>>
```

We can include multiple assignments on the same line. For example, the following expressions are valid

```
>> x = 2; y = 4; z = x*y
z =
      8
```

The Workspace Window shows the variables that have been created in the current Command Window and their values.

The following commands relate to variables:

1. ***who*** shows variables that have been defined in this Command Window (this just shows the names of the variables)

2. ***whos*** shows variables that have been defined in this Command Window (this shows more information on the variables, similar to what is in the Workspace Window)
3. ***clear*** clears out all variables so they no longer exist
4. ***clear variablename*** clears out a particular variable

If nothing appears when **who** or **whos** is entered, that means there aren't any variables! For example, in the beginning of a OCTAVE session, variables could be created and then selectively cleared (remember that the semicolon suppresses output):

```
>> who
>> mynum = 3;
>> mynum + 5;

>> who
Your variables are:
ans  mynum

>> clear mynum

>> who
Your variables are:
ans
```

### 3.1.3. Expressions

Expressions can be created using values, variables that have already been created, operators, built-in functions, and parentheses. For numbers, these can include operators such as multiplication, and functions such as trigonometric functions. An example of such an expression would be:

```
>> 2 * sin(1.4)
ans =
      1.9709
```

### 3.1.4. The FORMAT FUNCTION and ELLIPSIS

#### 3.1.4.1. ELLIPSIS

Long assignments can be extended to next line by typing an ellipsis which is just three periods in a row. For example:

```
>> FirstClassHolders = 72;
>> Coach = 121;
>> Crew = 8;
>> TotalPeopleOnPlane = FirstClassHolders + Coach...
+ Crew

TotalPeopleOnPlane =

   201
```

The ellipsis follows Coach on the line used to define TotalPeopleOnPlane. After you type the ellipsis, just hit the enter key. OCTAVE will move to the following line awaiting further input.

### 3.1.4.2. FORMAT FUNCTION

The ***default*** in OCTAVE is to display numbers that have decimal places with four decimal places, as already shown. The format command can be used to specify the output format of expressions. There are many options, including making the **format short** (the default) or **long** and **bank**. For example, changing the format to long will result in 15 decimal places. This will remain in effect until the format is changed back to short, as demonstrated with an expression and with the built-in value for ***pi***.

```
>> format long
>> 2 * sin(1.4)
ans =
     1.970899459976920

>> pi
ans =
     3.141592653589793

>> format short
>> 2 * sin(1.4)
ans =
     1.9709

>> pi
ans =
     3.1416
>> format bank
>> hourly = 35.55
```

The format command can also be used to control the spacing between the OCTAVE command or expression and the result; it can be either **loose** (the default) or **compact**.

```
>> format loose
>> 2^7

ans =

     128

>> format compact
>> 2^7
ans =
     128
```

OCTAVE displays large numbers using exponential notation. That is it represents $5.4387 \times 10^3$ as 5.4387e + 003. If you want all numbers to be represented in this fashion, you can do so. This type of notation can also be defined using the short (*format short* e) or long formats (*format long* e).

```
>> format short e
>> 7.2*3.1
ans =
   2.2320e+001
```

Also there is *format rat*. If you type *format rat*, then OCTAVE will find the closest rational expression it can that corresponds to the result of a calculation.

```
>> format rat
>> 7.2*3.1
ans =
     558/25
```

### 3.1.5. Basic Mathematical Definitions

OCTAVE is equipped with built-in functions (mathematical quantities) that we can use.

**Example 1.2**

1. Find the volume of a sphere of radius 2 m.

   The volume of a sphere is given by:

$$V = \frac{4}{3}\pi R^3$$

```
>> r = 2;
>> V = (4/3) *pi*r^3
V =
   33.5103
```

Another built-in function is the exponential function. That is, e ≈ 2.718. we can call this function e in OCTAVE by typing exp(a) which gives us the value of ea. Here are a few quick examples:

```
>> exp(1)
ans =
    2.7183

>> exp(2)
ans =
    7.3891
```

To find the square root of a number, we type sqrt. For example

```
>> x = sqrt(9)
x =
    3

>> y = sqrt(11)
y =
    3.3166
```

To find the natural *log* of a number say *x*, type *log(x)*

```
>> log(3.2)
ans =
    1.1632
>> x = 5; log(5)
ans =
    1.6094
```

To find the base ten logarithm, type *log10(x)*

```
>> x = 3; log10(x)
ans =
    0.4771
```

### 3.1.6. Built-In Functions and Help

There are many, many built-in functions in OCTAVE. The ***help*** command can be used to find out what functions OCTAVE has, and also how to use them. For instance, typing ***help*** at the prompt in the Command Window will show a list of help topics, which are groups of related functions. This is a very long list; the most elementary help topics are in the beginning.

For example, one of these is listed as **Octave\elfun;** it includes the elementary math functions. Another of the first help topics is **Octave\ops**, which shows the operators that can be used in expressions.

To see a list of the functions contained within a particular help topic, type help followed by the name of the topic.

For example,

```
>> help elfun
```

will show a list of the elementary math functions. It is a very long list, and is broken into trigonometric (for which the default is radians, but there are equivalent functions that instead use degrees), exponential, complex, and rounding and remainder functions.

To find out what a particular function does and how to call it, type help and then the name of the function.

For example,

```
>> help sin
```

will give a description of the **sin** function.

#### 3.1.6.1. Complex Numbers

We can also enter complex numbers in OCTAVE. Recall that the square root of –1 is defined as:

$$i = \sqrt{-1}$$

A complex number is one that can be written in the form $z = x + iy$, where $x$ the real is part of $z$ and $y$ is the imaginary part of $z$. It is easy to enter complex numbers in OCTAVE, by default it recognizes i as the square root of minus one.

In normal algebra it is written as:

```
a = 2 +3i
b = 1 − i
⇒ a + b = 3 + 2i
```

In OCTAVE it is written as:

```
>> format short
>> a = 2 + 3i;
>> b = 1 - i;
>> c = a + b
c =
   3.0000 + 2.0000i
```

### 3.1.7. Fixing Errors

It's going to be a fact that now and then you are going to type in an expression with an error. If you hit the enter key and then later realize what happened, it's not necessary to retype the line. Just use your arrow keys to move back up to the offending line. Fix your error, and then hit enter again and OCTAVE will correct the output.

### 3.1.8. File Basics

For instance, you want to save all the expressions and variables you have entered in the command window for use at a later time. You can do this by executing the following actions:

1. Click on the File pull-down menu
2. Select Save Workspace As…
3. Type in a file name
4. Click on the Save button

This method creates a OCTAVE file which has a **.MAT** file extension in Windows. If you save a file this way, you can retrieve all the commands in it and work with it again just like you can when working with fi les in any other computer program.

Sometimes, especially when working on complicated projects, you won't want to sit there and type every expression in a command window. It might be more appropriate to type a long sequence of operations and store them in a fi le that can be executed with a single command in the command window. This is done by creating a *script file*. This type of file is known as OCTAVE program and is saved in a file format with a **.M** extension. For this reason, we also call them **M-files**. We can also create M-files that are function files. From what we've done so far, you already know how to create a *script file*. All a *script file* comes down to is a saved sequence of OCTAVE commands. Let's create a simple script file that will compute $e^x$ for a few values of $x$. First, open the OCTAVE editor. Either

1. Click New → M-File under the File pull-down menu
2. Or click on the New File icon on our toolbar at the top of the screen

Now type in the following lines:

```
% script file example1.m to compute exponential of a set of numbers
x = [1:2:3:4];
y = exp(x)
```

Notice the % sign. This line is a comment. This is a line of text that is there for our benefit, it's a descriptive note that OCTAVE ignores. The next line creates an array or set of numbers. An array is denoted using square braces [] and by delimiting the elements of the array with colons or commas. The final line will tell OCTAVE to calculate the exponential of each member of the array, in other words the values $e^1$,$e^2$, $e^3$, $e^4$.

Save the file by clicking the Save icon in the fi le editor or by selecting Save As from the File pull-down menu. Save the file as example1.m in your OCTAVE directory.

Now return to the OCTAVE desktop command window. Type in example1. If you did everything right, then you will see the following output:

```
>> example1
y =
   2.7183      7.3891      20.0855      54.5982
```

We can also use M-fi les to create and store data. For instance, let's create a set of temperatures that we will store in a file. We do this by creating a list of temperatures in the file editor

```
temps = [32,50,65,70,85]
```

Save this as a fi le that we'll call TemperatureData.m. We store this file in the OCTAVE directory. To access it in the command window, we just type the name of the fi le. OCTAVE responds by spitting out the list of numbers:

```
>> TemperatureData
temps =
      32          50          65          70          85
```

Now we can use the data by referring to the array name used in the fi le. Let's create another set of numbers called Celsius that converts these familiar temps into the European style Celsius temperatures we are so familiar with. This can be done with the following command.

```
>> CelsiusTemps = (5/9) * (temps - 32)
CelsiusTemps =
   0     10.0000     18.3333     21.1111     29.4444
```

# Quiz

Use MATLAB to calculate the following quantities:

1. $5\frac{11}{14}$
2. $5\frac{8}{3}+3^7$
3. $9^{1.25}$
4. True or False. If $y$ has not been assigned a value, MATLAB will allow you to define the equation $x = y$ ^2 to store in memory for later use.
5. If the volume of a cylinder of height $h$ and radius $r$ is given by $V = \pi r^2 h$, use MATLAB to find the volume enclosed by a cylinder that is 12 cm high with a diameter of 4 cm.
6. Use MATLAB to compute the sin of $\pi/3$ expressed as a rational number.
7. Create a MATLAB m file to display the results of $\sin(\pi/4)$, $\sin(\pi/3)$, $\sin(\pi/2)$ as rational numbers.

## 4.0. Vectors and Matrices

## 4.1. Vectors

Recall that a vector is a *one-dimensional array of numbers (either row or column vector)*. Either column vectors or row vectors can be created using OCTAVE. A ***column vector*** can be created in OCTAVE by enclosing a set of ***semicolon*** delimited numbers in square brackets. Vectors can have any number of elements.

1. Column vector: To create a column vector with three elements we write:

```
>> a = [2; 1; 4]
a =
     2
     1
     4
```

Basic operations on column vectors can be executed by referencing the variable name used to create them. If we multiply a column vector by a number, this is called *scalar multiplication*. For example, to multiply vector **a** above by a number say **3** to produce another vector we write:

```
>> c = 3;
>> b = c*a
b =
     6
     3
    12
```

2. Row vector: To create a row vector, we enclose a set of numbers in square brackets but this time use a ***space*** or ***comma*** to delimit the numbers. For example

```
>> v = [1  2  3  4]
v =
   1   2   3   4
>> v = [1,2,3,4]
v =
   1   2   3   4
```

The ***transpose operation*** can be used to turn a column vectors into row vectors and vice versa. Suppose that we have a column vector with ***n*** elements denoted by:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

The transpose is given by

$$v^T = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$$

In OCTAVE, the transpose operation is represented with a single quote or tick mark (').
For instance, transpose of a column vector produces a row vector:

```
>> a = [2; 1; 4];
>> y = a'
y =
     2     1     4
```

And taking transpose of a row vector to produce a column vector

```
>> Q = [2 1 3]
Q =
     2     1     3
>> R = Q'
R =
     2
     1
     3
```

Mathematical operations such as addition or subtraction of two vectors can be done to produce another. However, to perform this operation the vectors must both be of the *same type and the same length*, so we can add two column vectors together to produce a new column vector:

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

Or we can subtract two row vectors to produce a new row vector:

```
>> W = [3,0,3];
>> X = [2,1,1];
>> Y = W - X
Y =
     1    -1     2
```

### 4.1.1. Generating Larger Vectors from Available Variables

In OCTAVE vectors can be attach together to create new ones. Let **u** and **v** be two column vectors with *m* and *n* elements respectively that we have created in OCTAVE. A third vector **w** can be created whose first *m* elements are the elements of **u** and whose next *n* elements are the elements of **v**. The newly created column vector has *m* + *n* elements. However, to perform this operation the vectors must both be of the *same type*. This is done by writing w = [u; v]. For example:

```
>> A = [1; 4; 5];
>> B = [2; 3; 3];
>> D = [A;B]
D =
     1
     4
     5
     2
     3
     3
```

This can also be done using row vectors:

```
>> R = [12, 11, 9]
>> S = [1, 4];
>> T = [R, S]
T =
    12    11     9     1     4
```

### 4.1.2. Generating Vectors with Uniformly Spaced Elements

It is possible to create a vector with elements that are uniformly spaced by an increment $\boldsymbol{q}$, where $\boldsymbol{q}$ is any real number. To create a vector $x$ with uniformly spaced elements where $x_i$ is the first element and $x_e$ is the final element, the syntax is:

$$x = [x_i : q : x_e]$$

For example, we can create a list of even numbers from 0 to 10 by writing:

```
>> x = [0:2:10]
x =
     0     2     4     6     8    10
```

This approach can also be used for some vectors with a small number of elements. First we create a set of $x$ values:

```
>> x = [0:0.1:1]
x =

   Columns 1 through 10

         0    0.1000    0.2000    0.3000    0.4000    0.5000
0.6000    0.7000    0.8000    0.9000

   Column 11

    1.0000
```

The set of $x$ values above can use to perform other operations. Suppose that $y = e_x$. Then we have:

```
>> y = exp(x)
y =

   Columns 1 through 10

    1.0000    1.1052    1.2214    1.3499    1.4918    1.6487
1.8221    2.0138    2.2255            2.4596

   Column 11

    2.7183
```

Or we have $y = x^2$:

```
      >> y = x. ^2
y =

   Columns 1 through 10

         0    0.0100    0.0400    0.0900    0.1600    0.2500
0.3600    0.4900    0.6400    0.8100

   Column 11

    1.0000
```

Note that when squaring a vector in OCTAVE, a period must precede the power operator (**. ^**). If we just enter >> y = x^2, OCTAVE gives us an error message:

```
??? Error using ==> mpower
Matrix must be square.
```

In furtherance to uniform increment, we can also use a negative increment. For instance, let's create a list of numbers from 100 to 80 decreasing by 5:

```
>> u = [100:-5:80]
u =
   100    95    90    85    80
```

#### 4.1.2.1. The linspace, logspace, length, max, min and abs commands

1. The linspace function creates a linearly spaced vector; ***linspace***$(x, y, n)$creates a vector with *n* values in the inclusive range from $x$ to $y$. For example, the following creates a vector with five values linearly spaced between 3 and 15, including the 3 and 15.

```
>> ls = linspace(3,15,5)
ls =
     3    6    9    12    15
```

However, linspace$(a, b)$ creates a row vector of 100 regularly spaced elements between $a$ and $b$.

Note that in both cases OCTAVE determines the increment in order to have the correct number of elements.

2. OCTAVE also allows you to create a row vector of $n$ logarithmically spaced elements by typing

```
logspace(a,b,n)
```

This creates $n$ regularly spaced elements between $10^a$ and $10^b$. For example:

```
>> logspace(1,2,5)
ans =
   10.0000   17.7828   31.6228   56.2341  100.0000
```

Or another example:

```
>> logspace(-1,1,6)
ans =
    0.1000    0.2512    0.6310    1.5849    3.9811   10.0000
```

3. The *length command* returns the number of elements that a vector contains. For example:

```
>> A = [2;3;3;4;5];
>> length(A)
ans =
     5
>> B = [1;1];
>> length(B)
ans =
     2
```

4. We can find the largest elements in a vector using the ***max*** command. For example:

```
>> A = [8 4 4 1 7 11 2 0];
>> max(A)
ans =
    11
```

5. We can find the smallest elements in a vector using the ***min*** command For example:

```
>> A = [8 4 4 1 7 11 2 0];
>> min(A)
ans =
     0
```

6. We can use the ***abs*** command to return the absolute value of a vector, which is a vector whose elements are the absolute values of the elements in the original vector, i.e.:

```
>> A = [-2 0 -1 9] >> B = abs(A)
B =
     2     0     1     9
```

#### 4.1.2.2. Vector Magnitude

To find the magnitude of a vector we can employ two operations. Recall that the magnitude of a vector ***v***

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Is given by

$$|v| = \sqrt{v_1^2 + v_2^2 + \cdots v_n^2}$$

To perform this operation, we will first take the dot product of a vector with itself. This is done by using *array multiplication* (.*). First let's define a vector:

```
>> J = [0; 3; 4];
```

Taking the array multiplication

```
>> J.*J
ans =
     0
     9
    16
```

This produces a vector whose elements are $v_1^2, v_2^2, \ldots$. To get the summation we need, we can use the sum operator:

```
>> a = sum(J.*J)
a =
    25
```

Then the magnitude of the vector is the square root of this quantity:

```
>> mag = sqrt(a)
mag =
     5
```

If a vector contains complex numbers, then more care must be taken when computing the magnitude. When computing the row vector, we must compute the complex conjugate transpose of the original vector. For example, if:

$$u = \begin{bmatrix} i \\ 1+2i \\ 4 \end{bmatrix}$$

Then to compute the magnitude, we need the following vector:

$$u^\dagger = [-i \quad 1-2i \quad 4]$$

Then the summation we need to compute is:

$$u^\dagger u = [-i \quad 1-2i \quad 4]\begin{bmatrix} i \\ 1+2i \\ 4 \end{bmatrix} = (-i)(i) + (1-2i)(1+2i) + (4)(4) = 22$$

Hence the magnitude of a vector with complex elements is:

$$|u| = \sqrt{u^{\dagger}u} = \sqrt{22}$$

From above its clear that using the complex conjugate transpose when computing our sum ensures that the magnitude of the vector will be real. Performing the same operation in OCTAVE, first we enter this column vector:

```
>> u = [i; 1+2i; 4];
```

We will compute the complex conjugate of the vector, and form the sum. We can get the complex conjugate of a vector with the *conj* command:

```
>> v = conj(u)
v =
          0 - 1.0000i
     1.0000 - 2.0000i
     4.0000
```

Now we obtain the correct answer, and can get the magnitude:

```
>> b = sum(v.*u)
b =
    22
>> magu = sqrt(b)
magu =
    4.6904
```

The above process can be done in one step:

```
>> c = sqrt(sum(conj(u).*u))
c =
    4.6904
```

### 4.1.3. Vector Dot and Cross Products

1. The dot product between two vectors $A = (a_1\ a_2\ \ldots\ a_n)$ and $B = (b_1 b_2\ \ldots\ b_n)$ is given by:

$$\boldsymbol{A \cdot B} = \sum_i \boldsymbol{a_i b_i}$$

In OCTAVE, the dot product of two vectors $a, b$ can be calculated using the $\mathrm{dot}(a, b)$ command. The dot product between two vectors is a scalar, i.e. it's just a number.

```
>> a = [1;4;7]; b = [2;-1;5];
>> c = dot(a,b)
c =
    33
```

The dot product can be used to calculate the magnitude of a vector. All that needs to be done is to pass the same vector to both arguments. Consider the vector $J$ above:

```
>> J = [0; 3; 4];
>> dot(J,J)
ans =
    25
```

We can calculate the magnitude of the vector this way

```
>> mag = sqrt(dot(J,J))
mag =
     5
```

The dot operation works correctly with vectors that contain complex elements:

```
>> u = [-i; 1 + i; 4 + 4*i];
>> dot(u,u)
ans =
    35
```

2. The cross product. To compute the cross product, *the vectors must be three dimensional*. For example:

```
>> A = [1 2 3]; B = [2 3 4];
>> C = cross(A,B)
C =
    -1     2    -1
```

### 4.1.4. Referencing Vector Components

There are several techniques that can be used to reference one or more of the components of a vector. The $ith$ component of a vector $v$ can be referenced by writing $v(i)$. For example:

```
>> A = [12; 17; -2; 0; 4; 4; 11; 19; 27];
>> A(2)
ans =
    17
>> A(8)
ans =
    19
```

Referencing the vector with a colon, such as $v(:)$; tells OCTAVE to list all of the components of the vector:

```
>> A(:)
ans =
    12
    17
    -2
     0
     4
     4
    11
    19
    27
```

We can also pick out a range of elements out of a vector. We can reference components four to six by writing $A\ (4{:}\ 6)$ and use these to create a new vector with three components:

```
>> v = A(4:6)
v =
      0
      4
      4
```

## 4.2. Matrices

A *matrix* is a two-dimensional array of numbers. To create a matrix in OCTAVE, we enter each row as a sequence of *comma* or *space delimited numbers*, and then use semicolons to mark the end of each row. For example, consider:

$$A = \begin{bmatrix} -1 & 6 \\ 7 & 11 \end{bmatrix}$$

This matrix is entered in OCTAVE using the following syntax:

```
>> A = [-1,6; 7, 11];
```

Or consider the matrix **B** below:

$$B = \begin{bmatrix} 2 & 0 & 1 \\ -1 & 7 & 4 \\ 3 & 0 & 1 \end{bmatrix}$$

```
>> B = [2,0,1;-1,7,4; 3,0,1]
```

Many of the previously discussed operations on vectors can equally be carried out on matrices. For instance, scalar multiplication can be carried out referencing the name of the matrix:

```
>> A = [-2 2; 4 1]
A =
    -2     2
     4     1
>> C = 2*A
C =
    -4     4
     8     2
```

Similarly, matrices of the same type and length can be added or subtracted.

```
>> A = [5 1; 0 9];
>> B = [2 -2; 1 1];
>> A + B
ans =
     7    -1
     1    10
>> A - B
ans =
     3     3
    -1     8
```

The transpose operation can be done in matrix. The transpose operation switches the rows and columns in a matrix, for example:

$$A=\begin{bmatrix}-1 & 2 & 4\\ 0 & 1 & 6\\ 2 & 7 & 1\end{bmatrix},\Rightarrow A^T=\begin{bmatrix}-1 & 0 & 2\\ 2 & 1 & 7\\ 4 & 6 & 1\end{bmatrix}$$

Just as we did in vectors, we use the same notation of transpose in matrix (‘):

```
>> A = [-1 2 0; 6 4 1]
A =
    -1     2     0
     6     4     1
>> B = A'
B =
    -1     6
     2     4
     0     1
```

If the matrix contains complex elements, the transpose operation will compute the *conjugates*:

```
>> C = [1+i, 4-i; 5+2*i, 3-3*i]
C =
   1.0000 + 1.0000i   4.0000 - 1.0000i
   5.0000 + 2.0000i   3.0000 - 3.0000i
>> D = C'
D =
   1.0000 - 1.0000i   5.0000 - 2.0000i
   4.0000 + 1.0000i   3.0000 + 3.0000i
```

However, to compute the transpose of a matrix with complex elements *without computing the conjugate*, we use (.’):

```
>> D = C.'
D =
   1.0000 + 1.0000i   5.0000 + 2.0000i
   4.0000 - 1.0000i   3.0000 - 3.0000i
```

We can perform *array multiplication*. Note that this is not matrix multiplication. We use the same notation used when multiplying two vectors together (.*). For example:

```
>> A = [12 3; -1 6]; B = [4 2; 9 1];
>> C = A.*B
C =
    48     6
    -9     6
```

What the operation $A.^{*}B$ does is it performs component by component multiplication. Theoretically it works as follows:

$$A.*B=\begin{pmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{pmatrix}\begin{pmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\end{pmatrix}=\begin{pmatrix}(a_{11})(b_{11}) & (a_{12})(b_{12})\\ (a_{21})(b_{21}) & (a_{22})(b_{22})\end{pmatrix}$$

### 4.2.1. Matrix Multiplication

Consider two matrices $A$ and $B$. If $A$ is an $m \times p$ matrix and $B$ is a $p \times n$ matrix, they can be multiplied together to produce an $m \times n$ matrix. To do this in OCTAVE, we leave out the period (.) and simply write $A * B$. However, if the dimensions of the two matrices are not the same, the operation will generate an error.

Let's consider two matrices:

$$A=\begin{pmatrix}2 & 1\\ 1 & 2\end{pmatrix},\quad B=\begin{pmatrix}3 & 4\\ 5 & 6\end{pmatrix}$$

These are both $2 \times 2$ matrices, so matrix multiplication is permissible. The array multiplication of $A$ and $B$ will give:

```
>> A = [2 1; 1 2]; B = [3 4; 5 6];
>> A.*B
ans =
     6     4
     5    12
```

The matrix multiplication will give:

```
>> A*B
ans =
    11    14
    13    16
```

Now we leave out the '.' character and execute matrix multiplication, which produces quite a different answer.

Consider:

$$A=\begin{pmatrix}1 & 4\\ 8 & 0\\ -1 & 3\end{pmatrix},\quad B=\begin{pmatrix}-1 & 7 & 4\\ 2 & 1 & -2\end{pmatrix}$$

The matrix $A$ is a $3 \times 2$ matrix, while $B$ is a $2 \times 3$ matrix. Since the number of columns of $A$ matches the number of rows of $B$, we can calculate the product $AB$. In OCTAVE:

```
>> A = [1 4; 8 0; -1 3]; B = [-1 7 4; 2 1 -2];
>> C = A*B
C =
     7    11    -4
    -8    56    32
     7    -4   -10
```

While matrix multiplication is possible in this case, array multiplication is not. To use array multiplication, both row and column dimensions must agree for each array.

### 4.2.2. Further Matrix Operations

OCTAVE also allows several operations on matrices. For example, OCTAVE allows you to add a scalar to an array (vector or matrix). This operation works by adding the value of the scalar to each component of the array. Here is how it works with a row vector:

```
>> A = [1 2 3 4];
>> b = 2;
>> C = b + A
C =
     3     4     5     6
```

We can also perform left and right division on an array. This works by matching component by component, so the arrays have to be of the same size. For example, we tell OCTAVE to perform array right division by typing (./ ):

```
A = [2 4 6 8]; B = [2 2 3 1];
>> C = A./B
C =
     1     2     2     8
```

Array left division is indicated by writing C = A.\B (this is the same as C = B./A):

```
>> C = A.\B
C =
    1.0000    0.5000    0.5000    0.1250
```

Basically any mathematical operation you can think of can be implemented in OCTAVE with arrays. For instance, we can square each of the elements:

```
>> B = [2 4; -1 6]
B =
     2     4
    -1     6
>> B.^2
ans =
     4    16
     1    36
```

### 4.2.3. Special Matrix

The identity matrix is a square matrix that has ones along the diagonal and zeros elsewhere. To create an n × n identity matrix, type the following OCTAVE command:

```
eye(n)
```

To create a $4\,x\,4$ identity matrix we write:

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

In addition, to create an n × n matrix of zeros, we type zeros$(n)$. We can also create a $m \times n$ matrix of zeros by typing zeros$(m, n)$. It is also possible to generate a matrix completely filled with $1's$. This is done by typing ones$(n)$ or ones$(m, n)$ to create $n \times n$ and $m \times n$ matrices filled with $1's$, respectively.

### 4.2.4. Referencing Matrix Elements

Individual elements and columns in a matrix can be referenced using OCTAVE. Consider the matrix:

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

We can pick out the element at row position m and column position n by typing $A(m, n)$. For example:

```
>> A(2,3)
ans =
     6
```

To reference all the elements in the $ith$ column we write $A(:, i)$. For example, we can pick out the second column of A:

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

To pick out the elements in the $ith$ through $jth$ columns we type$A(:, i:j)$. Here we return the second and third columns:

```
>> A(:,2:3)
ans =
     2     3
     5     6
     8     9
```

We can pick out pieces or submatrices as well. Continuing with the same matrix, to pick out the elements in the second and third rows that are also in the first and second columns, we write:

```
>> A(2:3,1:2)
ans =
     4     5
     7     8
```

We can change the value of matrix elements using these references as well. Let's change the element in row 1 and column 1 to –8:

```
>> A(1,1) = -8
A =
    -8     2     3
     4     5     6
     7     8     9
```

To create an empty array in OCTAVE, simply type an empty set of square braces []. This can be used to delete a row or column in a matrix. Let's delete the second row of A:

```
>> A(2,:)=[]
A =
    -8     2     3
     7     8     9
```

This has turned the formerly 3 × 3 matrix into a 2 × 3 matrix.
It's also possible to reference rows and columns in a matrix and use them to create new matrices. In this example, we copy the first row of $A$ four times to create a new matrix:

```
>> E = A([1,1,1,1],:)
E =
    -8     2     3
    -8     2     3
    -8     2     3
    -8     2     3
```

This example creates a matrix out of both rows of A:

```
>> F = A([1,2,1],:)
F =
    -8     2     3
     7     8     9
    -8     2     3
```

### 4.2.5. Finding Determinants and Solving Linear Systems

The determinant of a square matrix is a number. For a 2 × 2 matrix, the determinant is given by:

$$D = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

To calculate the determinant of a matrix A in OCTAVE, simply write $\det(A)$. Here is the determinant of a 2 × 2 matrix:

```
>> A = [1 3; 4 5];
>> det(A)
ans =
    -7
```

To find the determinant of a 4 × 4 matrix is also done the same way:

```
>> B = [3 -1 2 4; 0 2 1 8; -9 17 11 3; 1 2 3 -3];
>> det(B)
ans =
  -533
```

Determinants is an important to linear system of equation because it is use to determine if a linear system of equations has a solution. A linear system of equation as a solution if $det \neq 0$.

Consider the following set of equations:

$$5x + 2y - 9z = 44$$
$$-9x - 2y + 2z = 11$$
$$6x + 7y + 3z = 44$$

Recall that $\mathbf{Ax = b}$

To find a solution to a system of equations like this, we can use two steps. First we find the determinant of the coefficient matrix A, which in this case is:

$$A = \begin{pmatrix} 5 & 2 & -9 \\ -9 & -3 & 2 \\ 6 & 7 & 3 \end{pmatrix}$$

In OCTAVE, the determinant of the A matrix is:

```
>> A = [5 2 -9; -9 -3 2; 6 7 3]
>> det(A)
ans =
   368
```

Since the determinant is nonzero solution exists.

The coefficient matrix B

$$b = \begin{bmatrix} 44 \\ 11 \\ 5 \end{bmatrix}$$

This solution is the column vector:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

OCTAVE allows us to generate the solution readily using left division. First we create a column vector of the numbers on the right-hand side of the system. We find:

```
>> b = [44;11;5];
>> A\b
ans =
   -5.1250
    7.6902
   -6.0272
```

### 4.2.6. Finding the Rank of a Matrix

The rank of a matrix is a measure of the number of *linearly independent* rows or columns in the matrix. If a vector is linearly independent of a set of other vectors that means it cannot be written as a linear combination of them. Simple example:

$$u=\begin{pmatrix}1\\-2\end{pmatrix}, v=\begin{pmatrix}3\\-4\end{pmatrix}, w=\begin{pmatrix}5\\-6\end{pmatrix}$$

Looking at these column vectors we see that:

$$2u+v=w$$

Hence *w* is linearly dependent on *u* and *v*, since it can be written as a linear combination of them. On the other hand:

$$u=\begin{pmatrix}2\\0\\0\end{pmatrix}, v=\begin{pmatrix}0\\-1\\0\end{pmatrix}, w=\begin{pmatrix}0\\0\\7\end{pmatrix}$$

form a linearly independent set, since none of these vectors can be written as a linear combination of the other two.

Consider the matrix:

$$A=\begin{pmatrix}0 & 1 & 0 & 2\\0 & 2 & 0 & 4\end{pmatrix}$$

The second row of the matrix is clearly twice the first row of the matrix. Hence there is only one unique row and the rank of the matrix is 1. Let's check this in OCTAVE. We compute the rank in the following way:

```
>> A = [0 1 0 2; 0 2 0 4];
>> rank(A)
ans =
     1
```

Another example:

$$B=\begin{pmatrix}1 & 2 & 3\\3 & 0 & 9\\-1 & 2 & -3\end{pmatrix}$$

The third column is three times the first column:

$$\begin{matrix}3\\9\\-3\end{matrix} = 3\begin{pmatrix}1\\3\\-1\end{pmatrix}$$

Therefore, it's linearly dependent on the other two columns (add zero times the second column). The other two columns are linearly independent since there is no constant α such that:

$$\begin{pmatrix}2\\0\\2\end{pmatrix}=\alpha\begin{pmatrix}1\\3\\-1\end{pmatrix}$$

So we conclude that there are two linearly independent columns, and rank$(B) = 2$. Let's check it in OCTAVE:

```
>> B = [1 2 3; 3 0 9; -1 2 -3];
>> rank(B)
ans =
   2
```

Now let's consider the linear system of equations with *m* equations and *n* unknowns:

$$\mathbf{Ax = b}$$

The augmented matrix is formed by concatenating the vector **b** onto the matrix **A**:

$$\mathbf{[A\ \ b]}$$

A linear system of equation has a solution if and only if rank$(A)$ = rank$(A\ b)$ (i.e. the rank equals the number of unknowns in the equation). If the rank is equal to $n$, then the system has a unique solution.

If rank$(A)$ = rank$(A\ b)$ but the rank $< n$, (i.e. rank is equal but less than the number of unknowns in the equation) there are an infinite number of solutions. If we denote the rank by *r*, then *r* of the unknown variables can be expressed as linear combinations of *n* – *r* of the other variables.

The above facts can use these facts to analyze linear systems with relative ease. If the rank condition is met and the rank is equal to the number of unknowns, the solution can be computed by using left division.

Consider the system:

$$\begin{aligned} x - 2y + z &= 12 \\ 3x + 4y + 5z &= 20 \\ -2x + y + 7z &= 11 \end{aligned}$$

The coefficient matrix is:

$$A = \begin{pmatrix} 1 & -2 & 1 \\ 3 & 4 & 5 \\ -2 & 1 & 7 \end{pmatrix}$$

We also have:

$$b = \begin{pmatrix} 12 \\ 20 \\ 11 \end{pmatrix}$$

And the augmented matrix is:

$$\mathbf{(A\ b)} = \begin{pmatrix} 1 & -2 & 1 & 12 \\ 3 & 4 & 5 & 20 \\ -2 & 1 & 7 & 11 \end{pmatrix}$$

The first step is to enter these matrices in OCTAVE:

```
>> A = [1 -2 1; 3 4 5; -2 1 7] b = [12; 20; 11]
```

We can create the augmented matrix by using concatenation:

```
>> C = [A b]
C =
     1    -2     1    12
     3     4     5    20
    -2     1     7    11
```

Now let's check the rank of A:

```
>> rank(A)
ans =
     3
```

The rank of the augmented matrix is:

```
>> rank(C)
ans =
     3
```

Since the ranks are the same, a solution exists. We also note that the rank r satisfies r = n since there are three unknown variables. This means the solution is unique. We find it by left division:

```
>> x = A\b
x =
    4.3958
   -2.2292
    3.1458
```

### 4.2.7. Finding the Inverse of a Matrix and the Pseudoinverse

The inverse of a matrix *A* is denoted by $A^{-1}$ such that the following relationship is satisfied:

$$AA^{-1} = A^{-1}A = 1$$

Consider the following matrix equation:

$$Ax = b$$

If the inverse of *A* exists, then the solution can be readily written as:

$$x = A^{-1}b$$

The inverse of a matrix A can be calculated in OCTAVE by writing:

$$\text{inv}(A)$$

The inverse of a matrix does not always exist. In fact, we can use the determinant to determine whether or not the inverse exists. If $\det(A) = 0$, then the inverse does not exist and we say the matrix is *singular*.

1. Let's get started by calculating a few inverses just to see how easy this is to do in OCTAVE. Starting with a simple 2 × 2 matrix:

$$A = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$$

First we check for the determinant of the A matrix:

```
>> A = [2 3; 4 5]
A =
     2     3
     4     5
>> det(A)
ans =
    -2
```

Since det$(A)$ ≠ 0, we can fi nd the inverse. OCTAVE tells us that it is:

```
>> inv(A)
ans =
   -2.5000    1.5000
    2.0000   -1.0000
```

2. Let's consider a 4 × 4 case in OCTAVE.
   First we create the matrix:

```
>> S = [1 0 -1 2; 4 -2 -3 1; 0 2 -1 1; 0 0 9 8];
```

   Checking its determinant we find:

```
>> det(S)
ans =
  -108
```

   Since det(S) ≠ 0, the inverse must exist. OCTAVE spits it out for us:

```
>> T = inv(S)
T =
   -0.9259    0.4815    0.4815    0.1111
   -0.6296    0.1574    0.6574    0.0556
   -0.5926    0.1481    0.1481    0.1111
    0.6667   -0.1667   -0.1667         0
```

3. Now let's look at how we can solve a system of equations using the inverse. Consider:

$$3x - 2y = 5$$
$$6x - 2y = 2$$

   The coefficient matrix is:

```
>> A = [3 -2; 6 -2]
A =
     3    -2
     6    -2
```

   The vector **b** for the system **Ax = b** is:

```
>> b = [5;2]
b =
     5
     2
```

   First let's check the determinant of A to ensure that the inverse exists:

```
>> det(A)
ans =
     6
```

Since the inverse exists, we can generate the solution readily in OCTAVE:

```
>> x = inv(A)*b
x =
   -1.0000
   -4.0000
```

We can only use the method described above, multiplying by the inverse of the coefficient matrix to obtain a solution, if the coefficient matrix is ***square***. *This means for the system of equations, the number of equations equals the number of unknowns*.

If there are fewer equations than unknowns, the system is called *underdetermined*. This means that the system has an infinite number of solutions. This is because only some of the unknown variables can be determined. The variables that remain unknown can assume any value; hence there are an infinite number of solutions. We take a simple example:

$$x + 2y - z = 3$$
$$5y + z = 0$$

From above we know that:

$$z = -5y$$
$$x = 3 - 7y$$

In this system, while we can find values for two of the variables (x and z), the third variable y is undetermined. We can choose any value of y we like, and the system will have a solution.

Another case where an infinite number of solutions exist for a system of equations and unknowns is when $\det(A) = 0$.

Here the ***pseudoinverse*** comes in handy. This solution gives the minimum norm solution for real values of the variables. That is, the solution vector *x* is chosen to have the smallest norm such that the components of *x* are real.

Let's consider a linear system of equations:

$$3x + 2y - z = 7$$
$$4y + z = 2$$

Obviously this system has an infinite number of solutions. We enter the data:

```
>> A= [3 2 -1; 0 4 1]; b = [7;2];
>> C = [A b]
C =
     3     2    -1     7
     0     4     1     2
```

Computing the rank, we have:

```
>> rank(A)
ans =
     2
>> rank(C)
ans =
     2
```

Since these ranks are equal, a solution exists. We can have OCTAVE generate a solution using left division:

```
>> x = A\b
x =
    2.0000
    0.5000
         0
```

OCTAVE has generated a solution by setting one of the variables (z in this case) to zero. This is typically what it does in cases like these, if you try to generate a solution using left division. The solution is valid of course, but remember it only holds when z = 0, and z can be anything.
We can also solve the system using the pseudoinverse. We do this by typing:

```
>> x = pinv(A)*b
x =
    1.6667
    0.6667
   -0.6667
```

OCTAVE uses the Moore-Penrose pseudoinverse to calculate pinv.

### 4.2.8. Matrix Decompositions

In this section, we will take a look at LU decomposition and see how to use it to solve a linear system of equations in OCTAVE. We can find the LU decomposition of a matrix A by writing:

$$[L, U] = lu(A)$$

For example, let's find the LU decomposition of:

$$A = \begin{pmatrix} -1 & 2 & 0 \\ 4 & 1 & 8 \\ 2 & 7 & 1 \end{pmatrix}$$

We enter the matrix and find:

```
>> A = [-1 2 0; 4 1 8; 2 7 1];
>> [L, U] = lu(A)
L =
   -0.2500    0.3462    1.0000
    1.0000         0         0
    0.5000    1.0000         0
U =
    4.0000    1.0000    8.0000
         0    6.5000   -3.0000
         0         0    3.0385
```

We can use the LU decomposition to solve a linear system. Suppose that A was a coefficient matrix for a system with

$$b = \begin{pmatrix} 12 \\ -8 \\ 6 \end{pmatrix}$$

The solution can be generated with two left divisions:

$$x = U\backslash(L\backslash b)$$

We find:

```
>> x = U\(L\b)
x =
   -6.9367
    2.5316
    2.1519
```

Consider the system:

$$3x + 2y - 9z = -65$$
$$-9x + 5y + 2z = 16$$
$$6x + 7y + 3z = 5$$

```
>> A = [3 2 -9; -9 -5 2; 6 7 3]; b = [-65; 16; 5];
```

Now let's find the LU decomposition of A:

```
>> [L, U] = lu(A)
L =
   -0.3333    0.0909    1.0000
    1.0000         0         0
   -0.6667    1.0000         0
U =
   -9.0000   -5.0000    2.0000
         0    3.6667    4.3333
         0         0   -8.7273
```

Now we use these matrices together with left division to generate the solution:

```
>> x = U\(L\b)
x =
    2.0000
   -4.0000
    7.0000
```

# Quiz

1. Find the magnitude of the vector $\boldsymbol{A} = (-1 \quad 7 \quad 3 \quad 2)$.
2. Find the magnitude of the vector $\boldsymbol{A} = (-1 + i \quad 7i \quad 3 \quad -2-2i)$.
3. Consider the numbers 1, 2, 3. Enter these as components of a column vector and as components of a row vector.
4. Given **A** = [1; 2; 3]; B = [4; 5; 6];, find the array product of the two vectors.
5. What command would create a $5 \times 5$ matrix with ones on the diagonal and zeros everywhere else?
6. Consider the two matrices $A = \begin{pmatrix} 8 & 7 & 11 \\ 6 & 5 & -1 \\ 0 & 2 & -8 \end{pmatrix}, B = \begin{pmatrix} 2 & 1 & 2 \\ -1 & 6 & 4 \\ 2 & 2 & 2 \end{pmatrix}$ and compute their array product and matrix product.
7. Suppose that $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$. Use it to create $B = \begin{pmatrix} 7 & 8 & 9 \\ 7 & 8 & 9 \\ 4 & 5 & 6 \end{pmatrix}$
8. Find a solution to the following set of equations:

$$\begin{aligned} x + 2y + 3z &= 12 \\ -4x + y + 2z &= 13 \\ 9y - 8z &= -1 \end{aligned}$$

   What is the determinant of the coefficient matrix?
9. Does a solution to the following system exist? What is it?

$$\begin{aligned} x - 2y + 3z &= 1 \\ x + 4y + 3z &= 2 \\ 2x + 8y + z &= 3 \end{aligned}$$

10. Use LU decomposition to find a solution to the system:

$$\begin{aligned} x + 7y - 9z &= 12 \\ 2x - y + 4z &= 16 \\ x + y - 7z &= 16 \end{aligned}$$

## 5.0. Plotting and Graphics

## 5.1. Basic 2D Plotting

Plotting a function in OCTAVE involves the following three steps:

1. Define the function
2. Specify the range of values over which to plot the function
3. Call the OCTAVE plot(x, y) function

When specifying the range over which to plot the function, we must also tell OCTAVE what increment we want it to use to evaluate the function. Using smaller increments will result in plots with a smoother appearance. If the increment is smaller, OCTAVE will evaluate the function at more points. But it's generally not necessary to go that small.

To plot the function $y = cos(x)$ over the range $0 \leq x \leq 10$. to start, we want to define this interval and tell OCTAVE what increment to use. The interval is defined using square brackets [] that are filled in the following manner:

```
[ start : interval : end ]
```

1. For example, if we want to tell OCTAVE to plot over 0 ≤ x ≤ 10 with an interval of 0.1, we type:

```
[0:0.1:10]
```

To assign this range to a variable name, we use the assignment operator. We also do this to tell OCTAVE what the dependent variable is and what function we want to plot. Hence to plot $y = cos(x)$, we enter the following commands:

```
>> x = [0:0.1:10];
>> y = cos(x)
```

Notice that we ended each line with semicolons. Remember, this suppresses OCTAVE output.

```
>> plot(x, y)
```

After typing the plot command, hit the enter key. After a moment OCTAVE will open a new window on the screen with the caption Figure 1. The plot is found in this window. For the example we used, we obtain the plot shown in Figure 3-1.

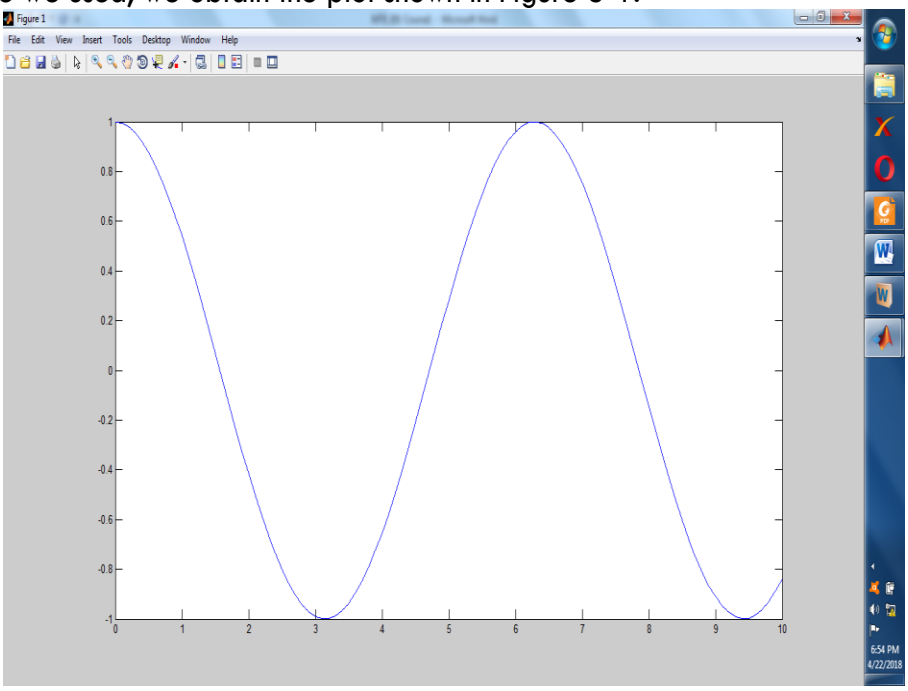


Figure 3-1 plot of $y = cos(x)$ generated by OCTAVE for 0 ≤ x ≤ 10

The next thing you might want to do is generate a plot that had the axes labeled. This can be done using the ***xlabel*** and ***ylabel*** functions. These functions can be used with a single argument, the label you want to use for each axis enclosed in quotes.
Place the ***xlabel*** and ***ylabel*** functions separated by commas on the same line as your plot command. For example, the following text generates the plot shown in Figure 3-2:

```
x=[0:0.01:10];
y=cos(x);
plot(x,y), xlabel('x'), ylabel('cos(x)');
```

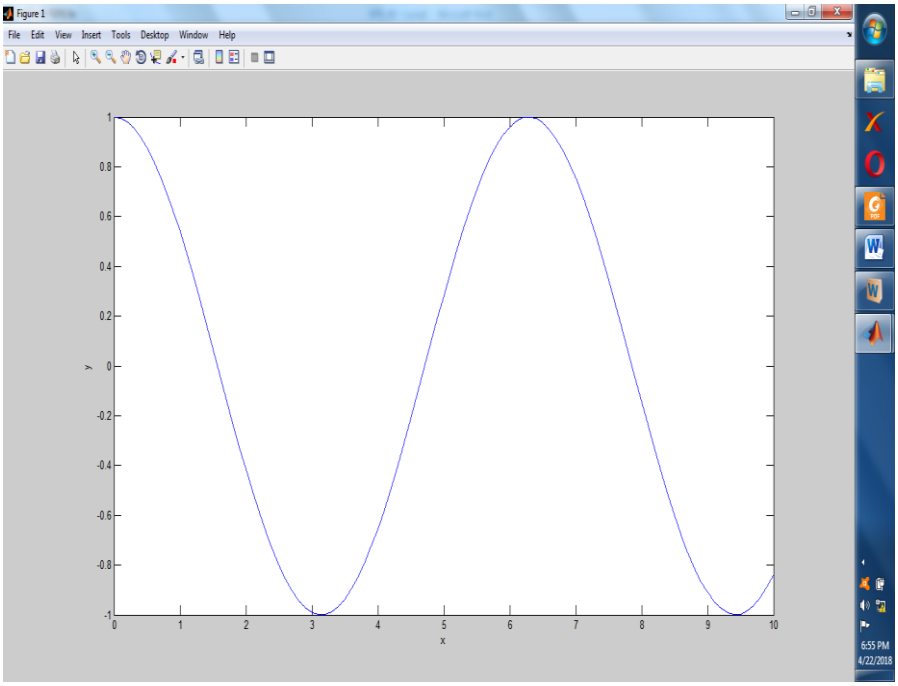



Figure 3-2 Sprucing up the plot with axis labels

2. Plot $y = tanh(x)$ over the range −6 ≤ x ≤ 6 with a grid display. First we define our interval:

```
>> x = [-6:0.01:6];
```

Next, we define the function:

```
>> y = tanh(x);
```

Now we will call the grid command with the plot command

```
Plot(x,y),grid on
```

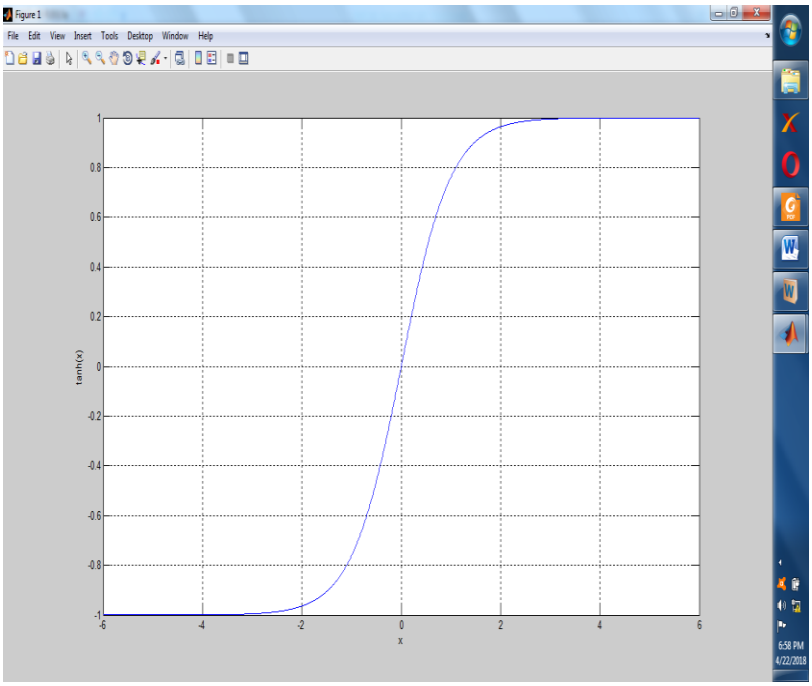


Figure 3-3 A plot made with the grid on command

### 5.1.1. The Axis Commands

The axes used in 2D plot can be adjusted in OCTAVE using the following commands:

i. If we add ***axis square*** (is the default plot axis type) to the line containing the ***plot*** command, then OCTAVE will generate a square plot.

ii. If we type ***axis equal***, then OCTAVE will generate a plot that has the same scale factors and tick spacing on both axes.

Using the $y = tanh(x)$ example, plotted in fig 3-3 above: If we run this plot with ***axis square***, we will get the same plot that we did using the default settings.

But suppose that we typed:

```
>> x = [-6:0.01:6];
>> y = tanh(x);
>> plot(x,y),grid on, axis equal
```

In this case, we get the plot shown in Figure 3-4. Notice that the spacing used for the y axis in Figure 3-3 and Figure 3-4 are quite different. In the first case, the spacing used on the vertical or y axis is different than the spacing used on the x axis.

In contrast, in Figure 3-4, the spacing is identical.

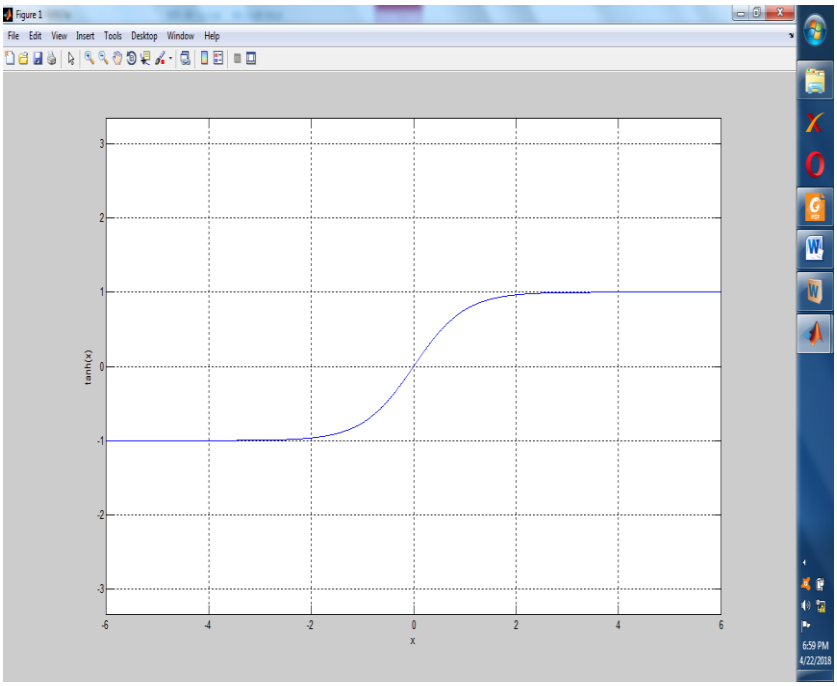


Figure 3-4 Plotting y = tanh(x) using the axis equal option

Obviously from this dramatic example, we can use the axis command to generate plots that differ quite a bit in appearance. Hence we can use the command to play with different plot styles and select what we need for the particular application.
To let OCTAVE set the axis limits automatically, type ***axis auto***. This isn't necessary, of course, unless you've been playing with the options described here.

### 5.1.2. Setting Axis Scales

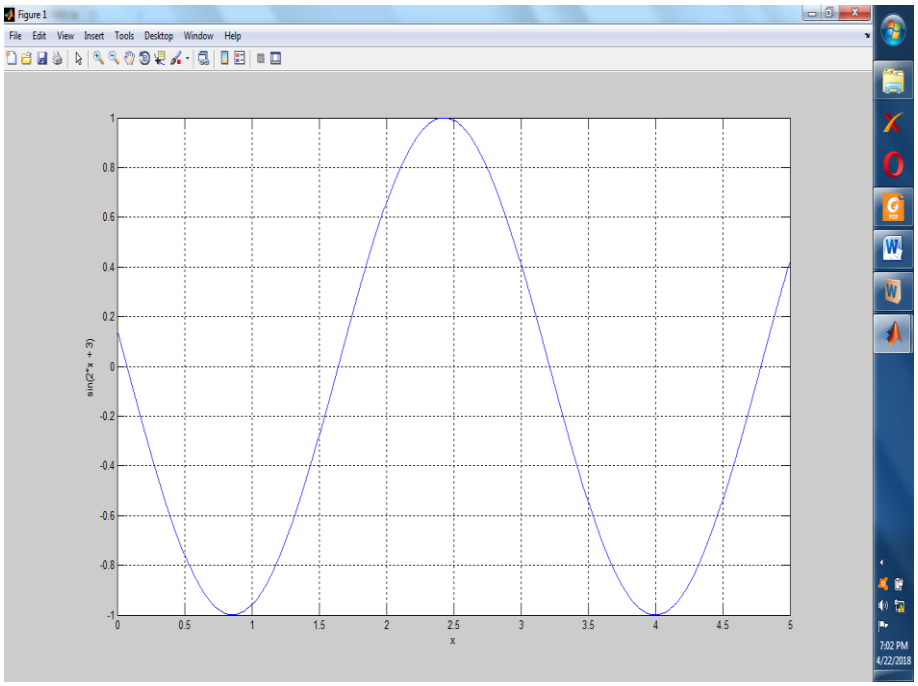
To set a plot range, this is done by calling ***axis*** command in the following way:

```
axis ( [xmin xmax ymin ymax] )
```

For instance, to generate a plot of $y = sin(2x + 3)$ for $0 \leq x \leq 5$ we might consider that the function ranges over $-1 \leq y \leq 1$. We can set the y axis to only show these values by using the following sequence of commands:

```
>> x = [0:0.01:5];
>> y = sin(2*x + 3);
>> plot(x,y), axis([0 5 -1 1])
```

This will generate the plot shown below:




A plot generated manually setting the limits on the
x and y axes for a plot of y = sin(2x + 3) for $0 \leq x \leq 5$

## 5.2. Showing Multiple Functions on One Plot

Often time, it is required to plot more than one curve on a single graph. The procedure used to do this in OCTAVE is fairly easy.
For instance, consider the functions:

$$f(t) = e^{-t}$$
$$g(t) = e^{-2t}$$

In this case let's plot the two functions over $0 \leq t \leq 5$

Step

1. Define the intervals:
```
>> t = [0:0.01:5];
```
2. Define the two functions:
```
>> f = exp(-t);
>> g = exp(-2*t);
```
3. Recall, to call the plot command we type *plot(x, y)*. To plot multiple functions, we simply call the ***plot(x, y)*** command with multiple pairs x, y defining the independent and dependent variables used in the plot in pairs. This is followed by a character string enclosed in single quotes to tell us what kind of line to use to generate the second curve. In this case we have:
```
>> plot(t,f,t,g,'--')
```

This tells OCTAVE to generate plots of $f(t)$ and $g(t)$ with the latter function displayed as a dashed line.

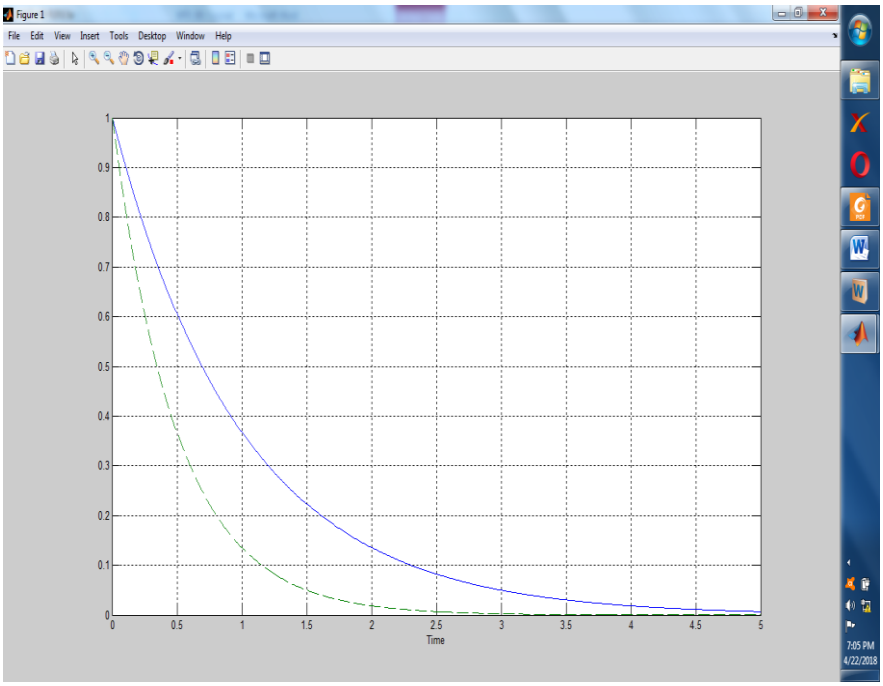


Figure 3-4 Plotting two curves on the same graph

OCTAVE has four basic line types that can be used to define a plot. These are, along with the character strings, used to define them in the plot command:

1. Solid line '-' (default)
2. Dashed line '--'
3. Dash-dot line '-.'
4. Dotted line ':'

We can generate same graph as in Figure 3-4 making the curve $f(t) = e^{-t}$ appear with a dotted line. The command is:

```
plot(t,f,':',t,g,'--')
```

This generates the plot shown in Figure 3-5

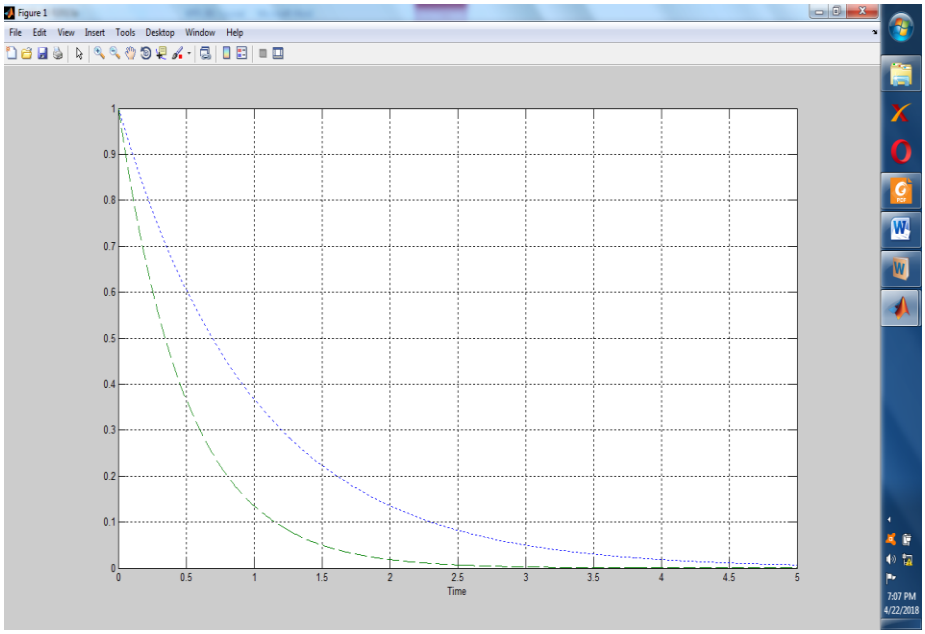



Figure 3-6 Using a dotted line to represent $f(t) = e^{-t}$ and a dashed line to represent $g(t) = e^{-2t}$

### 5.3. Adding Legends

A legend allows the reader of a plot to identify curves within the plot. For instance, let plot curve of two hyperbolic functions $sinh(x)$ and $cosh(x)$ for 0 ≤ x ≤ 2.

Step

1. First we define x:

```
>> x = [0:0.01:2];
```

2. Then, define the functions:

```
>> y = sinh(x);
>> z = cosh(x);
```

3. **Note:** The ***legend*** command is simple to use. Just add it to the line used for the *plot(x, y)* command and add a text string enclosed in single quotes for each curve you want to label. In our case we have:

```
legend('sinh(x)','cosh(x)')
```

We just add this to the plot command. For this example, we include x and y labels as well, and plot the curves using a solid line for the first curve and a dot-dash for the second curve:

```
>> plot(x,y,x,z,'-.'),xlabel('x'),ylabel('Potential'),legend('sinh(x)','cosh(x)')
```

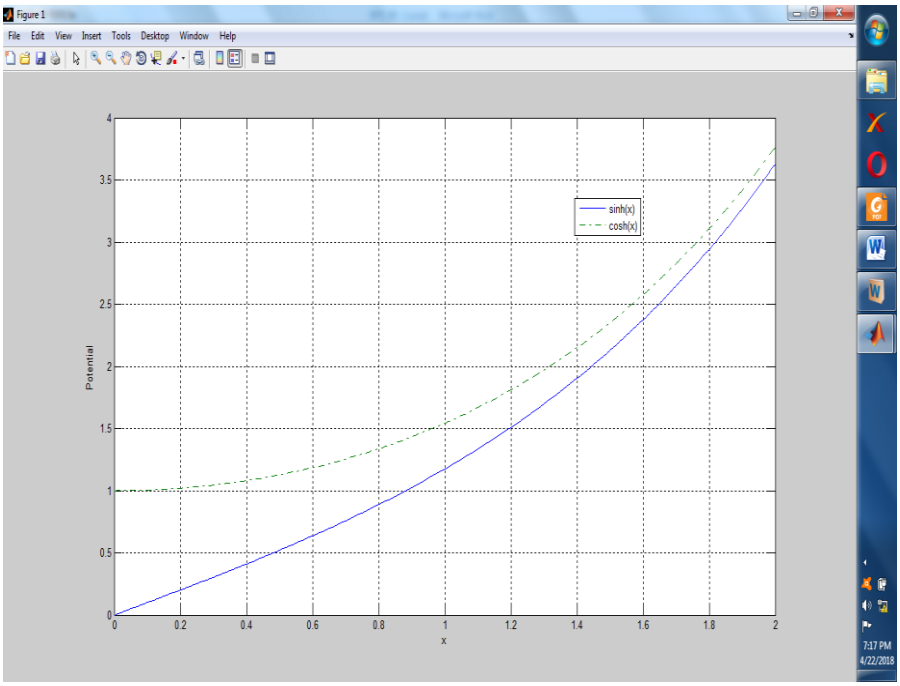



Figure 3-6 A plot of two curves that includes a legend

Note: the legend can be dragged and placed at any convenient place within the plot by just holding the mouse pointer over the legend and drag it to the location where you want it to display.

### 5.4. Setting Colors

The color of each curve can be set automatically by OCTAVE or we can manually select which color we want. This is done by enclosing the appropriate letter assigned to each color used by OCTAVE in single quotes immediately after the function to be plotted is specified.

For instance, let consider two hyperbolic functions $sinh(x)$ and $cosh(x)$ for -5 ≤ x ≤ 5.

Step

1. First we define x:

```
>> x = [-5:0.01:5];
```

2. Then, define the functions:

```
>> y = sinh(x);
>> z = cosh(x);
```

3. Now we will generate the plot representing y with a red curve and z with a blue curve. We do this by following our entries for y and z in the plot function by the character strings 'r' and 'b' respectively. The command looks like this:

```
>> plot(x,y,'r',x,z,'b')
```

In addition to the color specification we can equally specify the curve type alongside the color. So let's use the colors red and blue for the curves, and set the $cosh$ function (the blue curve) to draw with a dashed line.

```
>> plot(x,y,'r',x,z,'b--')
```

This gives us the plot shown in Figure 3-7

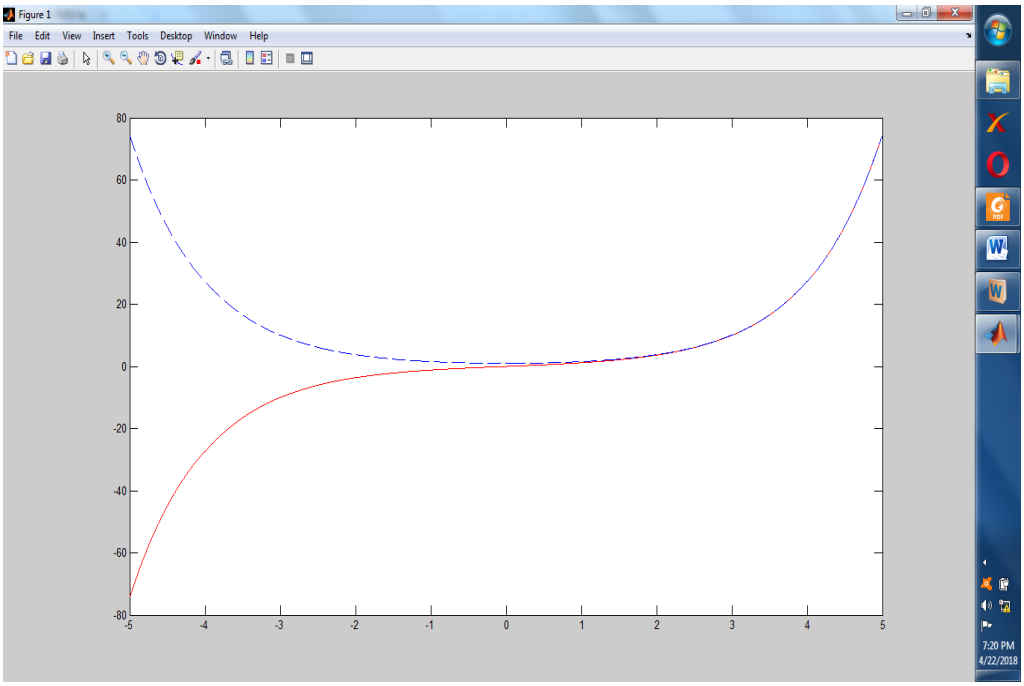



Figure 3-7 A plot generated setting colors and line types with the same command

OCTAVE gives the user eight basic color options for drawing curves. These are shown with their codes in Table 3-1.

Table 3-1 OCTAVE indicators for selecting plot colors.

| Color | Specifier |
|---|---|
| White | w |
| Black | k |
| Blue | b |
| Red | r |
| Cyan | c |
| Green | g |
| Magenta | m |
| Yellow | y |

The plot symbols, or markers, that can be used are:

```
o       circle
d       diamond
h       hexagram
p       pentagram
+       plus
.       point
s       square
*       star
v       down
        triangle
<       left
        triangle
>       right
        triangle
^       up triangle
x       x-mark
```

**Simple Related Plot Functions**

Other functions that are useful in customizing plots are ***clf***, ***figure***, ***hold***, ***legend***, and ***grid***. Brief descriptions of these functions are given here; you can use help to find out more about them:

1. **clf** clears the Figure Window by removing everything from it.
2. **figure** creates a new, empty Figure Window when called without any arguments. Calling it as **figure(n)** where **n** is an integer is a way of creating and maintaining multiple Figure Windows, and of referring to each individually.
3. **hold** is a toggle that freezes the current graph in the Figure Window, so that new plots will be superimposed on the current one. Just hold by itself is a toggle, so calling this function once turns the hold on, and then the next time turns it off. Alternatively, the commands hold on and hold off can be used.
4. **legend** displays strings passed to it in a legend box in the Figure Window, in order of the plots in the Figure Window.
5. **grid** displays grid lines on a graph. Called by itself, it is a toggle that turns the grid lines on and off. Alternatively, the commands grid on and grid off can be used.

## 5.5. Plotting Discrete Data

Plotting of graphs with discrete values of x and y is common place in practical analysis of data and OCTAVE also makes it possible to plot such data with relative ease. The $\boldsymbol{plot}\,(\boldsymbol{x},\boldsymbol{y})$ command can be used to plot a discrete set of data points and connect them with a line.

Consider the table below as an example:

| X axis | Y axis |
|---|---|
| 1 | 50 |
| 2 | 98 |
| 3 | 75 |
| 4 | 80 |
| 5 | 98 |

The task is for us to plot the data using OCTAVE.

Step

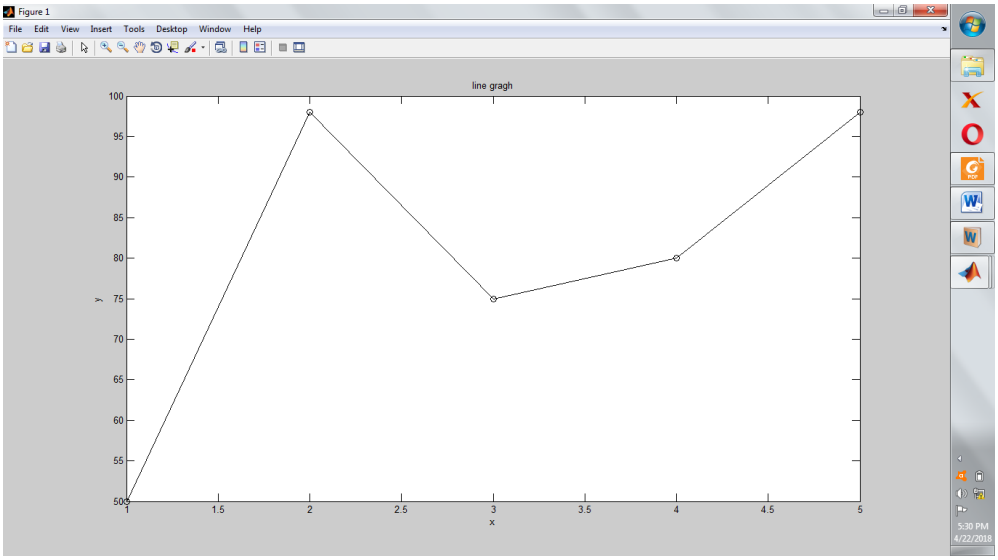
Define two arrays containing the list of students and the scores on the test.

```
>> x = [1:5];
```

Since we aren't modeling a continuous function, it's not necessary to specify an increment. So by default the increment is 1 and OCTAVE will generate 5 points. Next we put in the scores that correspond to each point; these are the y values, which we just create as a row vector.

```
>> y = [50,98,75,80,98];
```

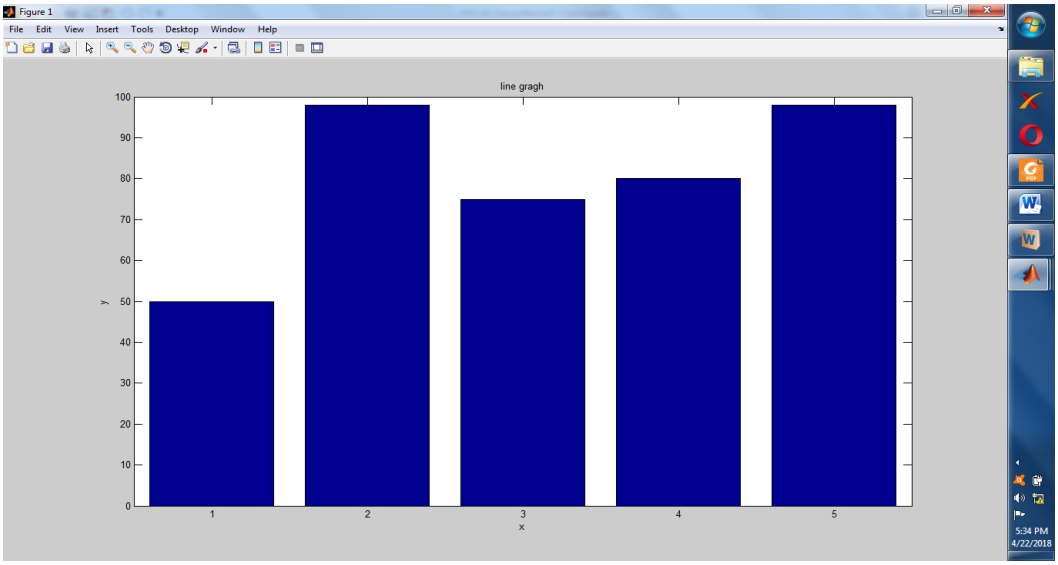
$$>> plot(x, y, ' ko', x, y, ' k'), axis\ auto, title('line\ gragh'), xlabel('x'), ylabel('y')$$




The same example can be used to plot a bar chart by calling the $\boldsymbol{bar}\ (\boldsymbol{x}, \boldsymbol{y})$ command

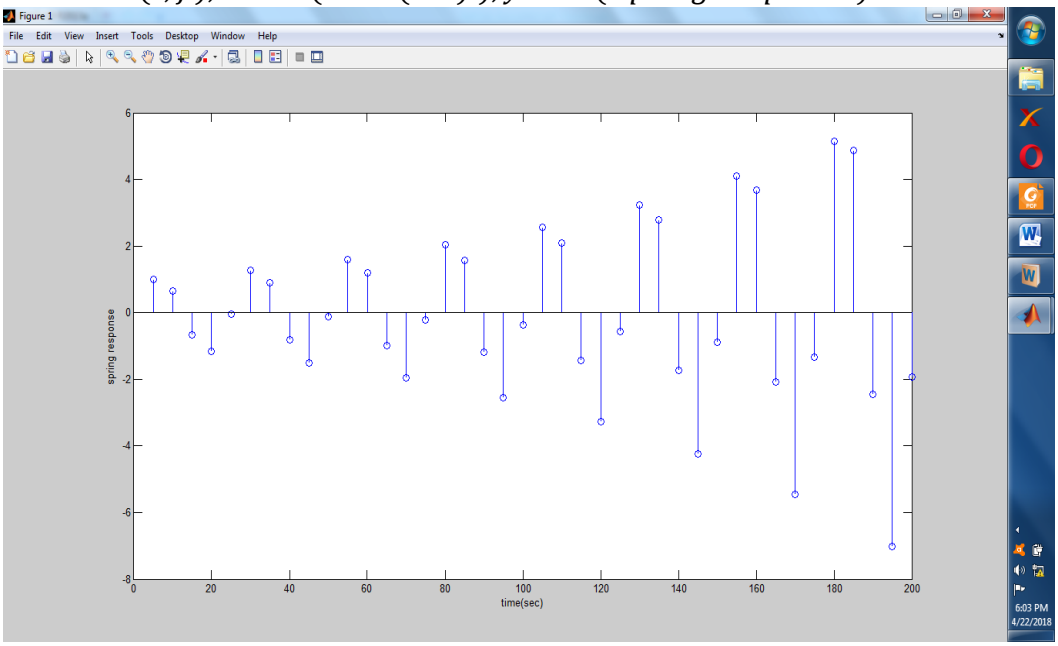
```
>> x = [1:5];
>> y = [50,98,75,80,98];>>
```

$$bar(x, y), title('line\ gragh'), xlabel('x'), ylabel('y')$$




In addition, OCTAVE also plot discrete data points using the ***stem plot*** command. The stem plot is a kind of discrete data plot used by engineers to show and analysis signals of some systems. The *stem plot* generates a graph of a function with data at certain discrete points. At each point, a vertical line extends from the horizontal or x axis up to the value of the function at that point, which is marked off with a selected marker. For instance, consider the function $f(t) = e^{\beta t}\sin(t/4)$ $\beta$ = 0.01 and assume it represents the response of a spring to some force. This response might have been produced experimentally or in a computer simulation. Suppose that the system was sampled every 5 seconds for 200 seconds. Assume we've got this data by first generating a set of sampling times. We simply create our array of times with a time step of 5 seconds:

$>> \ t = [0{:}\,5{:}\,200];$
$>> \ f = exp(0.01 * t).* sin(t/4);$
$>> \ stem(t, f), xlabel('time(sec)'), ylabel('spring\ response')$



## 5.6. Three-Dimensional (3D) Plots

OCTAVE has many functions that will display three-dimensional plots. Most of these functions have the same name as the corresponding two-dimensional ***plot*** function with a 3 at the end. For example, the three-dimensional line plot function is called ***plot3***. Other functions include ***bar3***, ***pie3***, and ***stem3***.

Vectors representing x, y, and z coordinates are passed to the ***plot3*** and ***stem3*** functions. These functions show the points in three-dimensional space. Clicking on the rotate 3D icon in the plot window allows the user to rotate the view to see the plot from different angles. Also, using the grid function makes it easier to visualize, as seen in Figure 3-8.

For instance, consider the program below:

```
>> x = [1:5];
>> y = [0 -2 4 11 3];
>> z = [2:2:10];
>> plot3(x,y,z,'k*')
>> grid on
```

This will generate the 3D plot below.

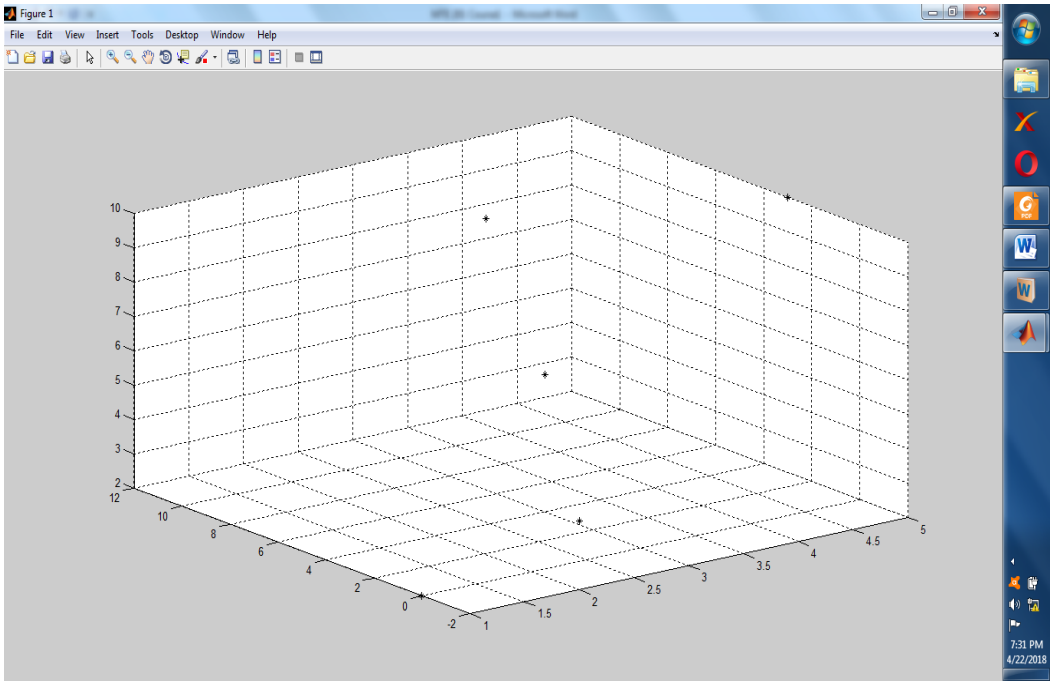



Figure 3-8 Three-dimensional plots with a grid on.

For the bar3 function, x and y vectors are passed and the function shows threedimensional bars as seen in Figure 3-9.

```
>> x = [1:6];
>> y = [33 11 5 9 22 30];
bar3(x,y);
```

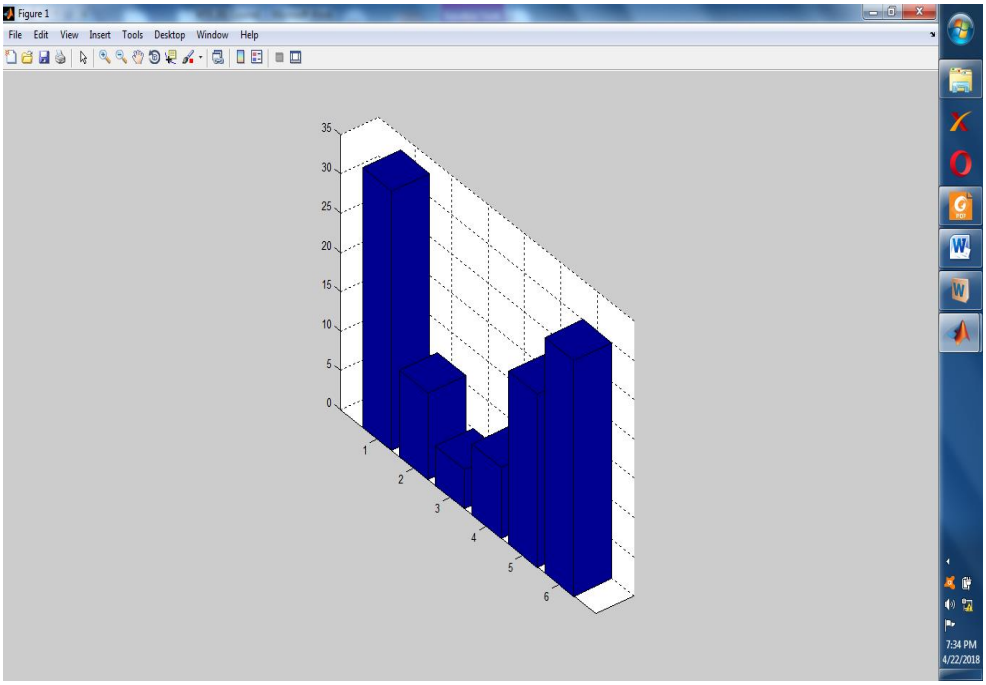



Figure 3-9 Three-dimensional bar chart

Similarly, the pie3 function shows data from a vector as a three-dimensional pie as seen in Figure 3-10.

$>> \ pie3([3\ 10\ 5\ 2])$

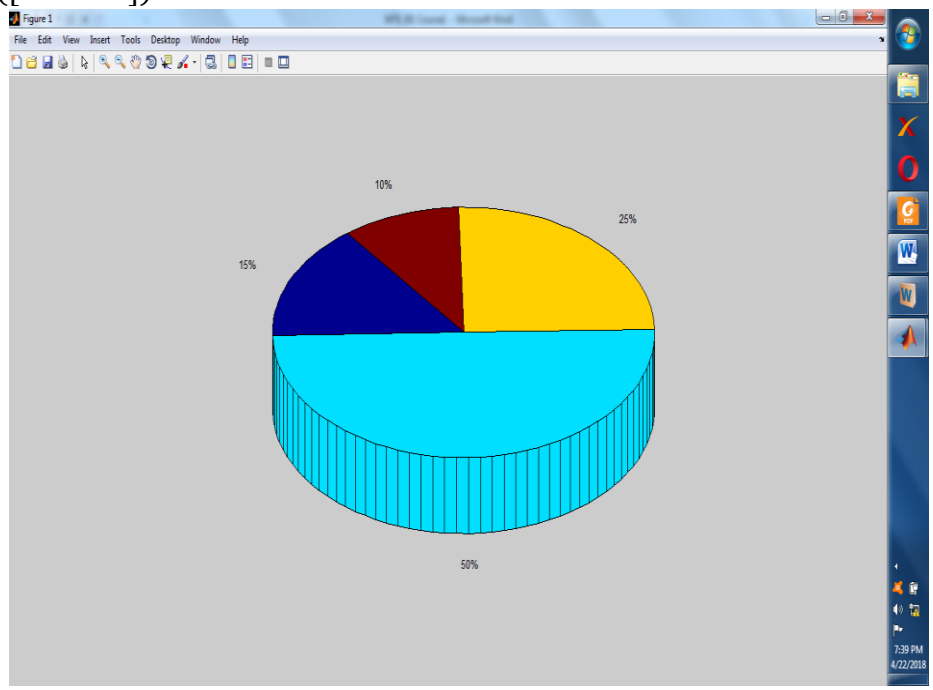



Figure 3-10 Three-dimensional pie charts.

The 3D plot can also be generated in OCTAVE calling the $\boldsymbol{mesh(x, y, z)}$ command. Consider the function $z = cos(x)sin(y)\ and\ -2\pi \leq x, y \leq 2\pi$. In this case the $\boldsymbol{meshgrid}$ command will specify the x and y axis.

We enter:

```
>> [x,y] = meshgrid(-2:0.1:2);
>> [x,y] = meshgrid(-2*pi:0.1:2*pi);
>> z = cos(x).*sin(y);
>> mesh(x,y,z),xlabel('x'),ylabel('y'),zlabel('z')
```

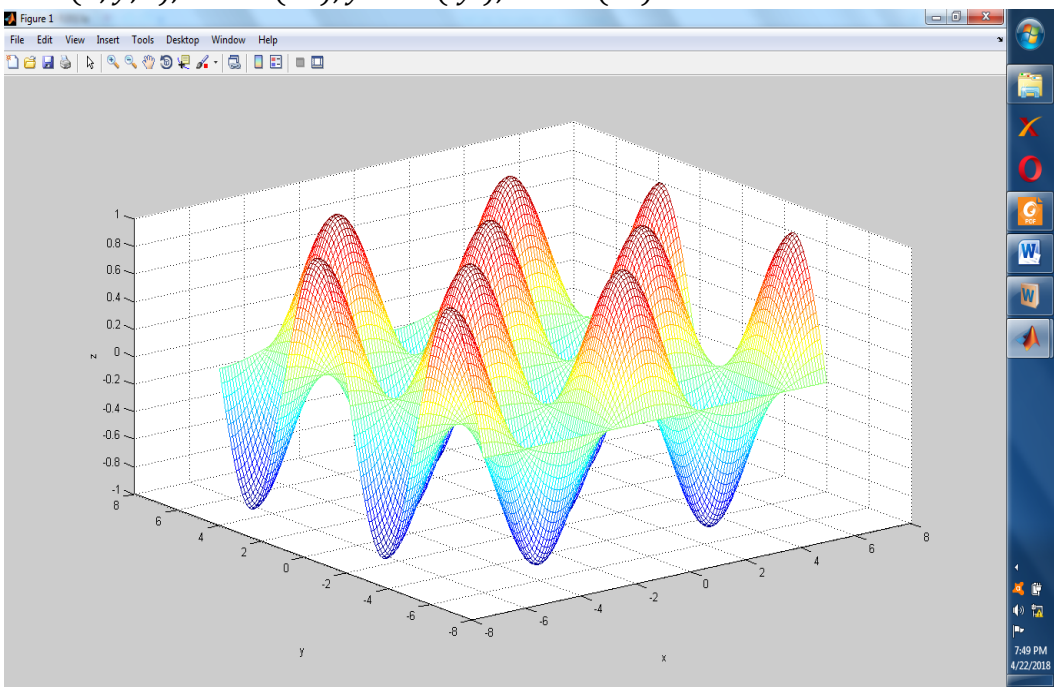



Figure 3-11 Plotting z = cos(x)sin(y) using the mesh command

We can also plot the function using shadeed surface plot. This is done by calling either the ***surf or surfc or surfl*** *command.*

```
>> [x,y] = meshgrid(-2:0.1:2);
>> [x,y] = meshgrid(-2*pi:0.1:2*pi);
>> z = cos(x).*sin(y);
>> surf(x,y,z),xlabel('x'),ylabel('y'),zlabel('z')
```

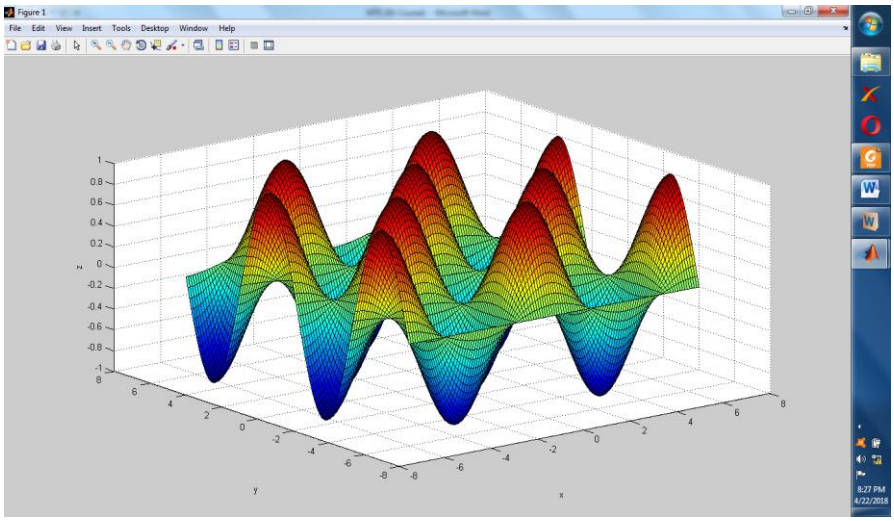


Figure 3-12 The same function plotted using surf(x, y, z)

Finally, s***urfl*** (the '**l**' tells us this is a lighted surface) is another command that generate nice option that gives the appearance of a three-dimensional illuminated object. Use this option if you would like a three-dimensional plot *without the mesh lines* shown in the other figures. Plots can be generated in color or grayscale. For instance, we use the following commands:

$>> \ [x, y] \ = \ meshgrid(-2\text{: } 0.1\text{: } 2);$

$>> \ [x, y] \ = \ meshgrid(-2 * pi\text{: } 0.1\text{: } 2 * pi);$

$>> \ z \ = \ cos(x).* sin(y);$

$>> \ surfl(x, y, z), xlabel('x'), ylabel('y'), zlabel('z')$

$>> \ shading \ interp$

$>> \ color \ map(gray);$

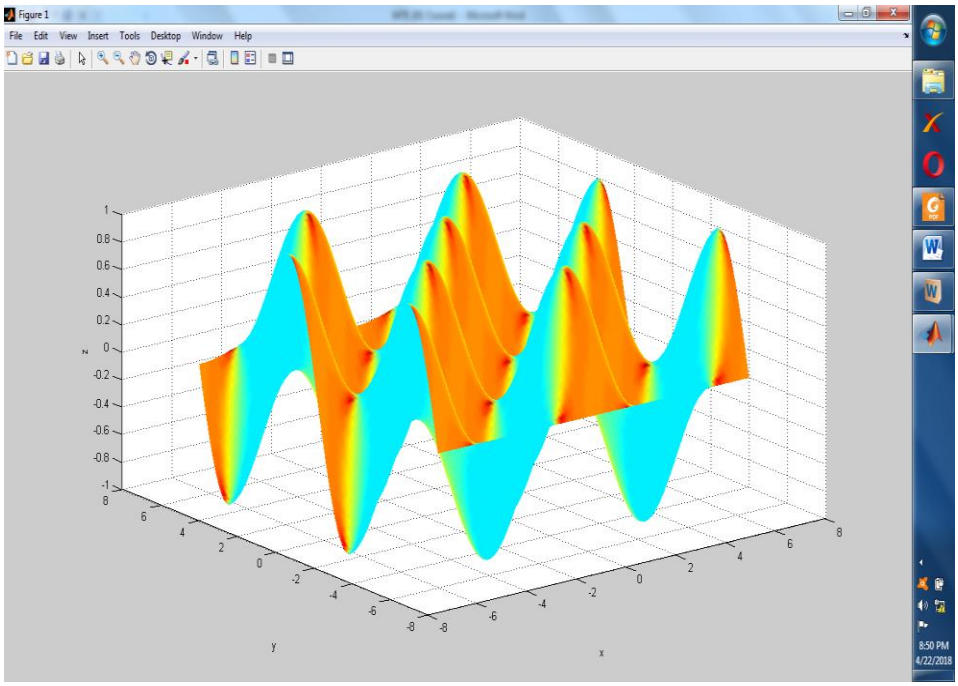



Figure 3-13 A plot of the function using surfl

This results in the impressive grayscale plot shown in Figure 3-13. The shading used in a plot can be set to *flat, interp, or faceted*.

1. flat shading assigns a constant color value over a mesh region with hidden mesh lines.
2. faceted shading adds the meshlines.
3. interp tells OCTAVE to interpolate what the color value should be at each point so that a continuously varying color map or grayscale shading scheme is generated, as we considered in Figure 3-13.