SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT – I – Programming in MATLAB – SEIA1503

UNIT 1 MATLAB PROGRAMMING

Introduction to MATLAB Software: MATLAB Command window - Workspace - Working with the MATLAB- user interface -Basic commands- Assigning variables Operations with variables-programming with different loops and conditionalstatements-Applications in Biomedical and Control engineering (signal & image).

MATLAB® is a very powerful software package that has many built-in tools for solving problems and for graphical illustrations. The simplest method for using the MATLAB product is interactively; an expression is entered by the user and MATLAB immediately responds with a result. It is also possible to write programs in MATLAB, which are essentially groups of commands that are executed sequentially. This chapter will focus on the basics, including many operators and built-in functions that can be used in interactive expressions. Means of storing values, including vectors and matrices, will also be introduced.

MATrix LABoratory (MATLAB)

- Basically deals with interactive matrix calculations
- Special purpose computer program optimized to perform Engineering and Scientific calculations
  - Has built in integrated development environment
- Supports different platform (windows 9x/NT/ 2000, Unix,etc.,)
- Has extensive library and built in functions for various field.
- MATLAB complier is an interpreter
- Includes tools that allow Graphical User Interface (GUI)

1.Getting Into Mat Lab

MATLAB 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. When the MATLAB software is started, a window is opened: the main part is the Command Window (see Figure 1.1). In the Command Window, there is a statement that says:
In the Command Window, you should see:
>>
The >> is called the prompt. In the Student Edition, the prompt
appears as: EDU>>

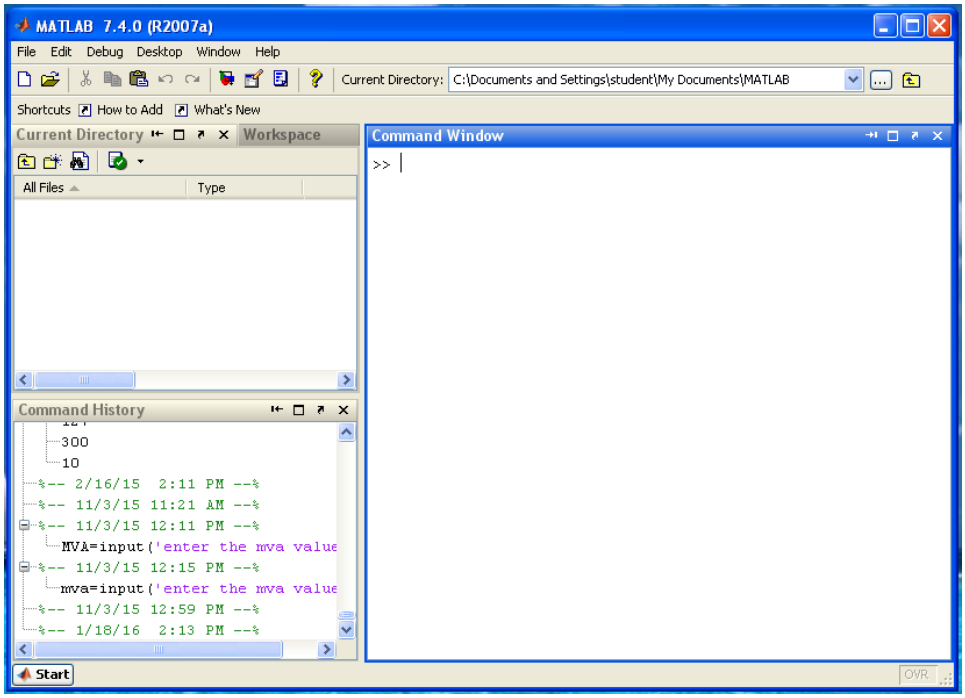


Figure 1.1 MATLAB Window

In the Command Window, MATLAB can be used interactively. At the prompt, any MATLAB command or expression can be entered, and MATLAB will immediately respond with the result. It is also possible to write ***programs*** in MATLAB, which are contained in ***script files*** or M-files. There are several commands that can serve as an introduction to MATLAB and allow you to get help:

info will display contact information for the product
demo has demos of several options in MATLAB
help will explain any command; **help help** will explain how help works

help browser opens a Help Window

lookfor searches through the help for a specific string (be aware that this can take a long time)

To get out of MATLAB, either type **quit** at the prompt, or chooses File, then Exit MATLAB from the menu. In addition to the Command Window, there are several other windows that can be opened and may be opened by default. What is described here is the default layout for these windows, although there are other possible configurations. Directly above the Command Window, there is a pull-down menu for the Current Directory. The folder that is set as the Current Directory is where files will be saved. By default, this is the Work Directory, but that can be changed.

To the left of the Command Window, there are two tabs for Current Directory Window and Workspace Window. If the Current Directory tab is chosen, the files stored in that directory are

displayed. The Command History Window shows commands that have been entered, not just in the current session (in the current Command Window), but previously as well. This default configuration can be altered by clicking Desktop, or using the icons at the top-right corner of

each window: either an -x,| which will close that particular window; or a curled arrow, which in its initial state pointing to the upper right lets you undock that window. Once undocked, clicking the curled arrow pointing to the lower right will dock the window again.

2.Variables And Assignment Statements

In order to store a value in a MATLAB session, or in a program, a ***variable*** is used. The Workspace Window shows variables that have been created. 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
>>

Here, the ***user*** (the person working in MATLAB) typed mynum = 6 at the prompt, and MATLAB stored the integer 6 in the variable called *mynum*, and then displayed the result followed by the prompt again. 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. Putting a semicolon at the end of a statement suppresses the output.
For example,
>> res = 9 – 2;

>>
This would assign the result of the expression on the right side, the value 7, to the variable *res*; it just doesn‘t show that result. Instead, another prompt appears immediately. However, at this point in the Workspace Window the variables *mynum* and *res* can be seen.
Note: In the remainder of the text, the prompt that appears after the result will not be shown. The spaces in a statement or expression do not affect the result, but make it easier to read. The following statement that has no spaces would accomplish exactly the same thing as the previous

statement:

>> res = 9–2;

MATLAB uses a default variable named *ans* if an expression is typed at the prompt and it is not assigned to a variable. For example, the result of the expression 6 3 is stored in the variable *ans*:
>> 6 + 3

ans
= 9
This default variable is reused any time just an expression is typed at the prompt. A short-cut for retyping commands is to press the up-arrow, which will go back to the previously typed command(s). For example, if you decided to assign the result of the expression 6 3 to the variable *res* instead of using the default *ans*, you could press the up-arrow and then the left-arrow to modify the command rather than retyping the whole statement:

>> res = 6 + 3

res
= 9
This is very useful, especially if a long expression is entered with an error, and you want to go back to correct it. 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

In the first assignment statement, the value 3 is assigned to the variable *mynum*. In the next assignment statement, *mynum* is changed to have the value of the expression 4 2, or 6. In the third assignment statement, *mynum* is changed again, to the result of the expression mynum 1. Since at that time *mynum* had the value 6, the value of the expression was 6 1, or 7. At that point, if the expression mynum 3 is entered, the default variable *ans* is used since the result of this expression is not assigned to a variable.

Thus, the value of *ans* becomes 10 but *mynum* is unchanged (it is still 7). Note that just typing the name of a variable will display its value.

>> mynum + 3

ans =
10
>>
mynum

mynum =
7

3.Initializing, Incrementing, And Decrementing

Frequently, values of variables change. Putting the first or initial value in a variable is called ***initializing*** the variable. Adding to a variable is called ***incrementing***. For example, the statement

mynum = mynum + 1

increments the variable *mynum* by 1.

VARIABLE NAMES

Variable names are an example of ***identifier names***. We will see other examples of identifier names, such as filenames, in future chapters. The rules for identifier names are:

- The name must begin with a letter of the alphabet. After that, the name can contain letters, digits, and the underscore character (e.g., value_1), but it cannot have a space.
- There is a limit to the length of the name; the built-in function **namelengthmax** tells how many characters this is.
- MATLAB is case-sensitive. That means that there is a difference between upper- and lowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different.
- There are certain words called ***reserved words*** that cannot be used as variable
- names. Names of built-in functions can, but should not, be used as variable names.

Additionally, variable names should always be ***mnemonic***, which means they should make some sense. For example, if the variable is storing the radius of a circle, a name such as -radius would make sense; probably wouldn't. The Workspace Window shows the variables that have been created in the current Command Window and their values.
The following commands relate to variables:

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

- **clear** clears out all variables so they no longer exist
  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 MATLAB 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

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 =
1.970
9

The Format **Function and Ellipsis**

The ***default*** in MATLAB 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**. 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 =
1.9708994599769
20
>>
pi
ans
=
3.141592653589793
>> format short

*>> 2 * sin(1.4)*

ans =
1.970
9
>> pi
ans =
3.141
6

The **format** command can also be used to control the spacing between the MATLAB 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
Especially long expressions can be continued on the next line by typing three (or more) periods, which is the continuation operator, or the **ellipsis**. For example,
>> 3 + 55 – 62 + 4 – 5 .â•›.â•›.

+ 22 – 1

ans
= 16

4.Built-In Functions And Help

There are many, many built-in functions in MATLAB. The **help** command can be used to find out what functions MATLAB has, and also how to use them. For example, 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 **matlab\elfun**; it includes the elementary math functions. Another of the first help topics is **matlab\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.
To ***call*** a function, the name of the function is given followed by the ***argument(s)*** that are passed to the function in parentheses. Most functions then ***return*** value(s). For example, to find the absolute value of –4, the following expression would be entered:

>> abs(–4)

which is a ***call*** to the function **abs**. The number in the parentheses, the –4, is the ***argument***. The value 4 would then be ***returned*** as a result. In addition to the trigonometric functions, the elfun help topic also has some rounding and remainder functions that are very useful. Some of these include **fix**, **floor**, **ceil**, **round**, **rem**, and **sign**. The **rem** function returns the remainder from a division; for example 5 goes into 13 twice with a remainder of 3, so the result of this expression is 3:

>> rem(13,5)

ans
= 3
Another function in the elfun help topic is the **sign** function, which returns 1 if the argument is positive, 0 if it is 0, and –1 if it is negative. For example,
>> sign(–5)

ans =
–1
>> sign(3)

ans
= 1

CONSTANTS

Variables are used to store values that can change, or that are not known ahead of time. Most languages also have the capacity to store ***constants***, which are values that are known ahead of time, and cannot possibly change. An example of a constant value would be **pi**, or , which is 3.14159…. In MATLAB, there are functions that return some of these constant values. Some of these include:

pi 3.14159….
i
j square root of 1
k square root of 1
inf infinity
NaN stands for -not a number; e.g., the result of 0/0

TYPES

Every expression, or variable, has a ***type*** associated with it. MATLAB supports many types of values, which are called ***classes***. A class is essentially a combination of a type and the operations that can be performed on values of that type. For example, there are types to store different kinds of numbers. For float or real numbers, or in other words numbers with a decimal place (e.g., 5.3), there are two basic types: **single** and **double**. The name of the type **double** is short for double precision; it stores larger numbers than **single**. MATLAB uses a ***floating point*** representation for these numbers. For integers, there are many integer types (e.g., **int8**, **int16**, **int32**, and **int64**). The numbers in the names represent the number of bits used to store values of that type. For example, the type **int8** uses eight bits altogether to store the integer and its sign. Since one bit is used for the sign, this means that seven bits are used to store the actual number. Each bit stores the number in binary (0‘s or 1‘s), and 0 is also a possible value, which means that 2 ^ 7 – 1 or 127 is the largest number that can be stored. The range of values that can be stored in **int8** is actually from –128 to 127. This range can be found for any type by passing the name of the type as a string (which means in single quotes) to the functions **intmin** and **intmax**. For example,

>> intmin(‘int8’)

ans =
–128
>> intmax(‘int8’)

ans
=
127

The larger the number in the type name, the larger the number that can be stored in it. We will for the most part use the type **int32** when an integer type is required. The type **char** is used to

store either single ***characters*** (e.g., _x‘) or ***strings***, which are sequences of characters (e.g., _cat‘). Both characters and strings are enclosed in single quotes. The type **logical** is used to store true/false values. If any variables have been created in the Command Window, they can be seen in the Workspace Window. In that window, for every variable, the variable name, value, and class (which is essentially its type) can be seen. Other attributes of variables can also be seen in the Workspace Window. Which attributes are visible by default depends on the version of MATLAB. However, when the Workspace Window is chosen, clicking View allows the user to choose which attributes will be displayed. By default, numbers are stored as the type **double** in MATLAB. There are, however, many functions that convert values from one type to another. The names of these functions are the same as the names of the types just shown. They can be used as functions to convert a value to that type. This is called ***casting*** the value to a different

type, or type casting. For example, to convert a value from the type **double**, which is the default, to the type **int32**, the function **int32** would be used. Typing the following assignment statement:

>> val = 6+3

would result in the number 9 being stored in the variable *val*, with the default type of **double**, which can be seen in the Workspace Window. Subsequently, the assignment statement

>> val = int32(val);

would change the type of the variable to **int32**, but would not change its value. If we instead stored the result in another variable, we could see the difference in the types by using **whos**.

>> val = 6 + 3;

>> vali = int32(val);

>> whos

Name Size Bytes Class Attributes
val 1x 8 doubl
1 e
vali 1x 4 int32
1

One reason for using an integer type for a variable is to save space.

RANDOM NUMBERS

When a program is being written to work with data, and the data is not yet available, it is often useful to test the program first by initializing the data variables to ***random numbers***. There are several built-in functions in MATLAB that ***generate*** random numbers, some of which will be illustrated in this section. Random number generators or functions are not truly random. Basically, the way it works is that the process. starts with one number, called a ***seed***. Frequently, the initial seed is either a predetermined value or it is obtained from the built-in clock in the computer. Then, based on this seed, a process determines the next random number. Using that number as the seed the next time, another random number is generated, and so forth. These are actually called ***pseudo-random***; they are not truly random because there is a process that determines the next value each time. The function **rand** can be used to generate random real numbers; calling it generates one random real number in the range from 0 to 1. There are no arguments passed to the **rand** function. Here are two examples of calling the **rand** function:

>> rand
ans =
0.9501

>> rand

ans =
0.2311

The seed for the **rand** function will always be the same each time MATLAB is started, unless the

state is changed, for example, by the following:

*rand('state',sum(100*clock))*

This uses the current date and time that are returned from the built-in **clock** function to set the seed. Note: this is done only once in any given MATLAB session to set the seed; the **rand** function can then be used as shown earlier any number of times to generate random numbers. Since **rand** returns a real number in the range from 0 to 1, multiplying the result by an integer N would return a random real number in the range from 0 to N. For example, multiplying by 10 returns a real in the range from 0 to 10, so this expression *rand*10*
would return a result in the range from 0 to 10. To generate a random real number in the range from *low* to *high*, first create the variables *low* and *high*. Then, use the expression rand*(high–low) low. For example, the sequence
>> low = 3;

>> high = 5;

>> rand(high–low)+low*

would generate a random real number in the range from 3 to 5.
However, in MATLAB, there is another built-in function that specifically generates random integers,
randint. Calling the function with **randint(1,1,N)** generates one random integer in the range from 0 to N –

1. The first two arguments essentially specify that one random integer will be returned; the third argument gives the range of that random integer. For example,

>> randint(1,1,4)

generates a random integer in the range from 0 to 3. Note: Even though this creates random integers, the type is actually the default type **double**. A range can also be passed to the **randint** function. For example, the following specifies a random integer in the range from 1 to 20:

>> randint(1,1,[1,20])

VECTORS AND MATRICES
Vectors and ***matrices*** are used to store sets of values, all of which are the same type. A vector can be either a ***row vector*** or a ***column vector***. A matrix can be visualized as a table of values.

The dimensions of a matrix are r × c, where r is the number of rows and c is the number of columns. This is pronounced -r by c.| If a vector has *n* elements, a row vector would have the dimensions 1 × *n*, and a column vector would have the dimensions *n* × 1. A ***scalar*** (one value) has the dimensions 1 × 1. Therefore, vectors and scalars are actually just subsets of matrices. Here are some diagrams showing, from left to right, a scalar, a column vector, a row vector, and a matrix:

5

3
7
4

5	88	3	11

9	6	3
5	7	2
4	33	8

The scalar is 1 × 1, the column vector is 3 × 1 (3 rows by 1 column), the row vector is 1 × 4 (1 row by 4 columns), and the matrix is 3 × 3. All the values stored in these matrices are stored in what are called ***elements***. MATLAB is written to work with matrices; the name MATLAB is short for -matrix laboratory.| For this reason, it is very easy to create vector and matrix variables, and there are many operations and functions that can be used on vectors and matrices. A vector in MATLAB is equivalent to what is called a one-dimensional ***array*** in other languages. A matrix is equivalent to a two-dimensional array. Usually, even in MATLAB, some operations that can be performed on either vectors or matrices are referred to as ***array operations***. The term ***array*** also frequently is used to mean generically either a vector or a matrix.

CREATING ROW VECTORS

There are several ways to create row vector variables. The most direct way is to put the values that you want in the vector in square brackets, separated by either spaces or commas. For example, both of these assignment statements create the same vector v:

>> v = [1 2 3 4]

v =

1 2 3 4

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

v =

1 2 3 4

Both of these create a row vector variable that has four elements; each value is stored in a separate element in the vector.

The Colon Operator and Linspace Function

If, as in the earlier examples, the values in the vector are regularly spaced, the ***colon operator*** can be used to ***iterate*** through these values. For example, 1:5
results in all the integers from 1 to 5:

>> vec = 1:5

vec =

1 2 3 4 5

Note that in this case, the brackets [ ] are not necessary to define the vector.
With the colon operator, a ***step value*** can also be specified with another colon, in the form (first:step:last). For example, to create a vector with all integers from 1 to 9 in steps of 2:

>> nv = 1:2:9

nv =

1 3 5 7 9

Similarly, 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

Vector variables can also be created using existing variables. For example, a new vector is created here consisting first of all the values from *nv* followed by all values from *ls*:

>> newvec = [nv ls]

newvec =

1 3 5 7 9 3 6 9 12 15

Putting two vectors together like this to create a new one is called ***concatenating*** the vectors.

REFERRING TO AND MODIFYING ELEMENTS

A particular element in a vector is accessed using the name of the vector variable and the element number (or ***index***, or ***subscript***) in parentheses. In MATLAB, the indices start at 1. Normally, diagrams of vectors and matrices show the indices; for example, for the variable *newvec* created earlier the indices 1–10 of the elements are shown above the vector:

1 2 3 4 5 6 7 8 9 10

1	3	5	7	9	3	6	9	12	15

For example, the fifth element in the vector *newvec* is a 9.

>> newvec(5)

ans
= 9

A subset of a vector, which would be a vector itself, can also be obtained using the colon operator. For example, the following statement would get the fourth through sixth elements of the vector *newvec*, and store the result in a vector variable *b*:

>> b = newvec(4:6)

b =
7 9 3

Any vector can be used for the indices in another vector, not just one created using the colon operator. For example, the following would get the first, fifth, and tenth elements of the vector *newvec*:

>> newvec([1 5 10])

ans =
1 9 15

The vector [1 5 10] is called an ***index vector***; it specifies the indices in the original vector that are being referenced. The value stored in a vector element can be changed by specifying the index or subscript. For example, to change the second element from the vector *b* to now store the value 11 instead of 9:

>> b(2) = 11

b =
7 11 3

By using an index, a vector can also be extended. For example, the following creates a vector that has three elements. By then referring to the fourth element in an assignment statement, the vector is extended to have four elements.

>> rv = [3 55 11]

rv =
3 55 11
>> rv(4) = 2

rv =
3 55 11 2

If there is a gap between the end of the vector and the specified element, 0‘s are filled in. For example, the following extends the variable created earlier again:
>> rv(6) = 13

rv =
3 55 11 2 0 13

CREATING COLUMN VECTORS
One way to create a column vector is by explicitly putting the values in square brackets, separated by semicolons:
>> c = [1; 2; 3; 4]

c
=
1
2
3
4

There is no direct way to use the colon operator described earlier to get a column vector. However, any row vector created using any of these methods can be ***transposed*** to get a column vector. In general, the transpose of a matrix is a new matrix in which the rows and columns are interchanged. For vectors, transposing a row vector results in a column vector, and transposing a column vector results in a row vector. MATLAB has a built-in operator, the apostrophe, to get a transpose.

>> r = 1:3;

>> c = r

c
=
1
2
3

CREATING MATRIX VARIABLES
Creating a matrix variable is really just a generalization of creating row and column vector

variables. That is, the values within a row are separated by either spaces or commas, and the different rows are separated by semicolons. For example, the matrix variable *mat* is created by explicitly typing values:

>> mat = [4 3 1; 2 5 6]

mat =

4	3	1
2	5	6

There must always be the same number of values in each row. If you attempt to create a matrix in which there are different numbers of values in the rows, the result will be an error message;

for example:

>> mat = [3 5 7; 1 2]

??? Error using ==> vertcat

CAT arguments dimensions are not consistent. Iterators can also be used for the values on the rows using the colon operator; for example:

>> mat = [2:4; 3:5]

mat =
2 3 4
3 4 5

Different rows in the matrix can also be specified by pressing the Enter key after each row instead of typing a semicolon when entering the matrix values; for example:

>> newmat = [2 6 88

33 5 2]

newmat =
2 6 88
33 5 2

Matrices of random numbers can be created using the **rand** and **randint** functions. The first two arguments to the **randint** function specify the size of the matrix of random integers. For example, the following will create a 2 × 4 matrix of random integers, each in the range from 10 to 30:

>> randint(2,4,[10,30])

ans =
29 22 28 19
14 20 26 10

```
>>
rand(1,3)
ans =
0.7621    0.4565    0.0185
```

MATLAB also has several functions that create special matrices. For example, the **zeros** function creates a matrix of all zeros. Like **rand**, either one argument can be passed (which will be both the number of rows and columns), or two arguments (first the number of rows and then the number of columns).

```
>> zeros(3)

ans =
     0     0     0
     0     0     0
     0     0     0

>> zeros(2,4)

ans =
     0     0     0     0
     0     0     0     0
```

Referring To And Modifying Matrix Elements

To refer to matrix elements, the row and then the column indices are given in parentheses (always the row index first and then the column). For example, this creates a matrix variable *mat*, and then refers to the value in the second row, third column of *mat*:

```
>> mat = [2:4; 3:5]

mat =
     2     3     4
     3     4     5

>> mat(2,3)

ans
= 5
```

It is also possible to refer to a subset of a matrix. For example, this refers to the first and second rows, second and third columns:

```
>> mat(1:2,2:3)
```

```
ans =
3     4
```

4 5

Using a colon for the row index means all rows, regardless of how many, and using a colon for the column index means all columns. For example, this refers to the entire first row:

```
>> mat(1,:)

ans =
2 3 4
```

and this refers to the entire second column:

```
>> mat(:, 2)

ans
= 3
4
```

If a single index is used with a matrix, MATLAB ***unwinds*** the matrix column by column. For example, for the matrix *intmat* created here, the first two elements are from the first column, and the last two are from the second column:

```
>> intmat = randint(2,2,[0 100])

intmat =
100 77
28 14

>> intmat(1)

ans
=
100
>> intmat(2)

ans
= 28
>> intmat(3)

ans
= 77
>> intmat(4)

ans
= 14
```

This is called ***linear indexing***. It is usually much better style when working with matrices to refer to the row and column indices, however. An individual element in a matrix can be modified

>> mat = [2:4; 3:5];

>> mat(1,2) = 11

mat =
2 11 4
3 4 5

An entire row or column could also be changed. For example, the following replaces the entire second row with values from a vector:
>> mat(2,:) = 5:7

mat =
2 11 4
5 6 7

Notice that since the entire row is being modified, a vector with the correct length must be assigned. To extend a matrix, an individual element could not be added since that would mean there would no longer be the same number of values in every row. However, an entire row or column could be added. For example, the following would add a fourth column to the matrix:

>> mat(:,4) = [9 2]'

mat =
2 11 4 9
5 6 7 2

Just as we saw with vectors, if there is a gap between the current matrix and the row or column being added, MATLAB will fill in with zeros.

>> mat(4,:) = 2:2:8

mat =
2 11 4 9
5 6 7 2
0 0 0 0
2 4 6 8

DIMENSIONS

The **length** and **size** functions in MATLAB are used to find array dimensions. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a matrix. For a matrix, the **length** function will return either the number of rows or the number of columns, whichever is largest. For example, the following vector, *vec*, has

```
vec =
-2      -1      0       1
>> length(vec)

ans
= 4
>> size(vec)

ans =
1       4
```

For the matrix *mat* shown next, it has three rows and two columns, so the size is 3 × 2. The length is the larger dimension, 3.

```
>> mat = [1:3; 5:7]'

mat =
1       5
2       6
3       7
>> size(mat)

ans =
3       2
>> length(mat)

ans
= 3
>> [r c] = size(mat)

r
=
3
c =
2
```

Note: The last example demonstrates a very important and unique concept in MATLAB: the ability to have a vector of variables on the left-hand side of an assignment. The **size** function returns two values, so in order to capture these values in separate variables we put a vector of two variables on the left of the assignment. The variable *r* stores the first value returned, which is the number of rows, and *c* stores the number of columns.

MATLAB also has a function, **numel**, which returns the total number of elements in any array (vector or matrix):

```
vec =
9 7 5 3 1

>> numel(vec)

ans
= 5

>> mat = randint(2,3,[1,10])

mat =
7        9        8
4        6        5

>> numel(mat)

ans
= 6
```

For vectors, this is equivalent to the length of the vector. For matrices, it is the product of the number of rows and columns. MATLAB also has a built-in expression **end** that can be used to refer to the last element in a vector; for example, v(end) is equivalent to v(length(v)). For matrices, it can refer to the last row or column. So, using **end** for the row index would refer to the last row. In this case, the element referred to is in the first column of the last row:

```
>> mat = [1:3; 4:6]'

mat =
1        4
2        5
3        6

>> mat(end,1)

ans
= 3
```

Using **end** for the column index would refer to the last column (e.g., the last column of the second row):

```
>> mat(2,end)

ans
= 5
```

Changing Dimensions

In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices, including **reshape**, **fliplr**, **flipud**, and **rot90**. The **reshape** function changes the dimensions of a matrix. The following matrix variable *mat* is 3 4, or in other words it has 12 elements.

>> mat = randint(3,4,[1 100])

mat =

14	61	2	94
21	28	75	47
20	20	45	42

These 12 values instead could be arranged as a 2 x 6 matrix, 6 x 2, 4 x3, 1x 12, or 12 x1. The **reshape** function iterates through the matrix columnwise. For example, when reshaping *mat* into a 2 6 matrix, the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

>> reshape(mat,2,6)

ans =

14	20	28	2	45	47
21	61	20	75	94	42

The **fliplr** function -flips| the matrix from left to right (in other words the left-most column, the first column, becomes the last column and so forth), and the **flipud** functions flips up to down. Note that in these examples *mat* is unchanged; instead, the results are stored in the default variable *ans* each time.

>> mat = randint(3,4,[1 100])

mat =

14	61	2	94
21	28	75	47
20	20	45	42

>>

fliplr(mat)

ans =

94	2	61	14
47	75	28	21
42	45	20	20

```
>>
mat
mat =  61     2     94
14
21     28    75    47
20     20    45    42

>> flipud(mat)

ans =
20     20    45    42
21     28    75    47
14     61    2     94
```

The **rot90** function rotates the matrix counterclockwise 90 degrees, so for example the value in the top-right corner becomes instead the top-left corner and the last column becomes the first row:

```
>> mat
mat =
14     61    2     94
21     28    75    47
20     20    45    42

>>
rot90(mat)
ans =          42
94     47
2      75    45
61     28    20
14     21    20
```

The function **repmat** can also be used to create a matrix; **repmat(mat,m,n)** creates a larger matrix, which consists of an $m \times n$ matrix of copies of mat. For example, here is a 2×2 random matrix:

```
>> intmat = randint(2,2,[0 100])

intmat =
100
       7
7
```

The function **repmat** can be used to replicate this matrix six times as a 3 × 2 matrix of the variable *intmat*.

>> repmat(intmat,3,2)

ans =

100	77	100	77
28	14	28	14
100	77	100	77
28	14	28	14
28	14	28	14

EMPTY VECTORS

An ***empty vector***, or, in other words, a vector that stores no values, can be created using empty square brackets:

>> evec = [ ]

evec
= [ ]
>> length(evec)

ans
= 0

Then, values can be added to the vector by ***concatenating***, or adding values to the existing vector. The following statement takes what is currently in *evec*, which is nothing, and adds a 4 to it.

>> evec = [evec 4]

evec
= 4

The following statement takes what is currently in *evec*, which is 4,
and adds an 11 to it.

>> evec = [evec 11]

evec =
4 11

This can be continued as many times as desired, in order to build a vector up from nothing. Empty vectors can also be used to ***delete elements from arrays***. For example, to remove the third element from an array, the empty vector is assigned to it:

>> vec = 1:5

vec =

1	2	3	4	5

1 2 4 5

The elements in this vector are now numbered 1 through 4. Subsets of a vector could also be removed; for example:

>> vec = 1:8

vec =

1 2 3 4 5 6 7 8

>> vec(2:4) = [ ]

vec =

1 5 6 7 8

Individual elements cannot be removed from matrices, since matrices always have to have the same number of elements in every row.

>> mat = [7 9 8; 4 6 5]

mat =

7 9 8

4 6 5

>> mat(1,2) = [ ];

??? Indexed empty matrix assignment is not allowed. However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

>> mat(:,2) = [ ]

mat =

7 8

4 5

CELL ARRAYS

It is a special MATLAB array whose elements are cells, container that can hold other MATLAB arrays

Cell array contains data structures instead of data

C{1,1} =

1 2 3

4 5 6

7 8 3

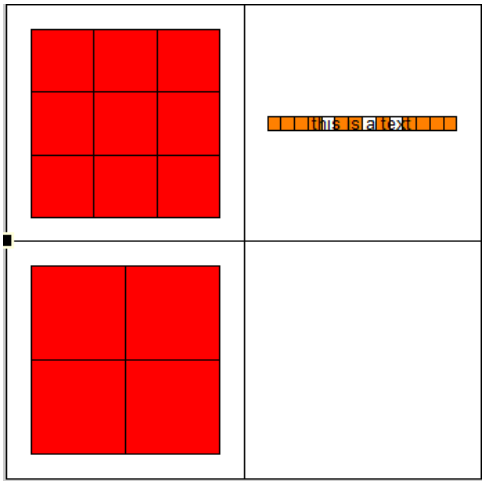
C{2,1} =

C{1,2} =

this is a text

C{2,2} =

[]



TO CREATE CELL ARRAY

Using assignment statement

Assignment with content indexing

C{1,1} = [1 2 3; 4 5 6; 7 8 9];

C{1,2} = ‘ This is a text’;

C{2,1} = [3+4*I -5; 2 -10*i];

C{2,2} = [ ];

Assignment with cell indexing

C(1,1) = {[1 2 3; 4 5 6; 7 8 9]};

C(1,2) = {‘ This is a text’};

C(2,1) = {[3+4*I -5; 2 -10*i]};

C(2,2) = {[ ]};

Pre-allocating cell array with cell function

B=cell(2,3)

```
>> B=cell(2,3)
B =
   []    []    []
   []    []    []
```

Use { } as cell constructor

Individual cell values can be created separated by comma

B = {[1 2],17,[2;3];3-4*I, ‘hello’, eye(3)}

```
>> B = {[1 2],17,[2;3];3-4*i, 'hello', eye(3)}
B =
        [1x2 double]    [   17]    [2x1 double]
   [3.0000 - 4.0000i]    'hello'    [3x3 double]
```

To View Content Of Cell Array

```
>> B
B =
        [1x2 double]    [   17]    [2x1 double]
   [3.0000 - 4.0000i]  'hello'    [3x3 double]
 >> celldisp(B)
 B{1,1} =
      1     2
B{2,1} =
   3.0000 - 4.0000i
 B{1,2} =
     17
 B{2,2} =
 Hello
```

2

3

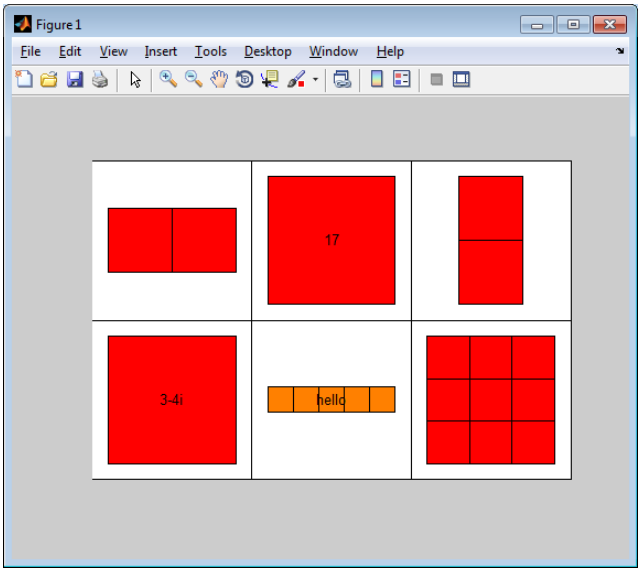
B{2,3} =

1 0 0

0 1 0

0 0 1

>> cellplot(B)



TO EXTENT CELL ARRAY

The existing cell array can be extended by assignment statement

>> B{4,4}=5

B =

[1x2 double] [17] [2x1 double] []

[3.0000 - 4.0000i] 'hello' [3x3 double] []

[] [] [] []

[] [] [] [5]

TO DELETE THE CONTENT OF CELL ARRAY

>> B(4,:) = [ ]

```
[3.0000 - 4.0000i]    'hello'    [3x3 double]    []

            []        []            []    []
```

The fourth row is deleted

There are several methods of displaying cell arrays. The **celldisp** function displays all elements of the cell array:

```
>> celldisp(cellrowvec)

cellrowvec{1
} = 23
cellrowvec{2
} = a
cellrowvec{3} =
1        3        5        7        9
cellrowvec{4
} = hello
```

The function **cellplot** puts a graphical display of the cell array in a Figure Window; however, it is a high- level view and basically just displays the same information as typing the name of the variable (e.g., it wouldn‘t show the contents of the vector in the previous example). Many of the functions and operations on arrays that we have already seen also work with cell arrays. For example, here are some related to dimensioning:

```
>> length(cellrowvec)

ans
 = 4
>> size(cellcolvec)

ans =
4        1
>> cellrowvec{end}

ans
=
hello
```

It is not possible to delete an individual element from a cell array. For example, assigning an empty vector to a cell array element does not delete the element, it just replaces it with the emptyvector:

```
>> cellrowvec
```

>> length(cellrowvec)

ans
= 4
>> cellrowvec{2} = [ ]

mycell =
[23] â [] [1x5 double] _hello‘

>> length(cellrowvec)

ans
= 4
However, it is possible to delete an entire row or column from a cell array by assigning the empty vector (**Note**: use parentheses rather than curly braces to refer to the row or column):

>> cellmat

mycellmat =
[23] _a‘
[1x5 double] _hello‘
>> cellmat(1,:) = []

mycellmat =
[1x5 double] _hello‘

Storing Strings in Cell Arrays
One good application of a cell array is to store strings of different lengths. Since cell arrays can store different types of values in the elements, that means strings of different lengths can be stored in the elements.

>> names = {‘Sue’, ‘Cathy’, ‘Xavier’}

names =
_Sue‘ _Cathy‘ _Xavier‘
This is extremely useful, because unlike vectors of strings created using **char** or **strvcat**, these strings do not have extra trailing blanks. The length of each string can be displayed using a **for** loop to loop through the elements of the cell array:

>> for i =
1:length(names)
disp(length(names{i}))
end

It is possible to convert from a cell array of strings to a character array, and vice versa. MATLAB has several functions that facilitate this. For example, the function **cellstr** converts from a character array padded with blanks to a cell array in which the trailing blanks have been removed.

>> greetmat = char('Hello','Goodbye');

>> cellgreets = cellstr(greetmat)

cellgreets =
'Hello'
'Goodbye'

The **char** function can convert from a cell array to a character matrix:
>> names = {'Sue', 'Cathy', 'Xavier'};

>> cnames = char(names)

cnames
= Sue
Cathy
Xavier

>> size(cnames)

ans =
3 6

The function **iscellstr** will return logical true if a cell array is a cell array of all strings, or logical false if not.

>> iscellstr(names)

ans
= 1

>> iscellstr(cellcolvec)

ans
= 0
We will see several examples of cell arrays containing strings of varying lengths in the coming chapters, including advanced file input functions and customizing plots.

Structure Array

A cell array is a data type in which there is a single name for the whole data structure.

A Structure is a data type in which each individual element has a name. The individual elements of a structure are known as fields.

Create Structure Array

A Field at a time using assignment statement

Assignment Statement

```
>> student.name='ram';

>> student.regno='3513110';

>> student.add='1st street';

>> student.city='Chennai';

>> student.zip='600119';
```

These assignment statement will create a structure named student with fields – name, regno,add,city,zip

To add another database

```
>> student(2).name='shiva';
```

Using struct function

```
>> student_database=struct('name', 'sathya','regno', [3513120])

student_database =

     name: 'sathya'

    regno: 3513120
```

Will create a structure named student_database with fields name and regno

```
>> student2_database(1000)=struct('name',[ ],'regno',[ ],'add',[ ])

student2_database =

1x1000 struct array with fields:

    name

    regno

    add
```

Will create a structure named student2_database with fields name, regno and add

TO ADD FIELDS TO STRUCTURE

>> student(2).mark = [88 80 76 90 78 81 99]

student =

1x2 struct array with fields:

name

regno

add

city

zip

mark

The field mark is added to the structure named student

TO REMOVE FIELDS TO STRUCTURE

```
>> student = rmfield(student,'zip')
```

student =

1x2 struct array with fields:

name

regno

add

city

mark

The field ‘zip’ is removed from the structure named student

TO EXTRACT DATA FROM STRUCTURE ARRAY

To get the information in the structure

```
>> student(2).add
```

ans =

[]

Returns empty array, since there is no data added to the field for student(2)

```
>> student(2).mark
```

ans =

88 80 76 90 78 81 99

Returns the marks corresponding to student(2)

>> student(2).mark(2)

ans =

80

Returns the mark corresponding to student(2) with index 2

>> mean(student(2).mark)

ans =

84.5714

Returns the mean of the marks corresponding to student(2)

Similarly any operation can be performed with the extracted data

TO EXTRACT/SET DATA

getfield is a function that gets the current value stored in the field

>> city= getfield(student(1),'city')

city =

Chennai

Returns the current value of the field ‘city’ corresponding to student(1)

setfield is a function that inserts new value in to the field

>> setfield(student(2),'regno',3513105')

ans =

name: 'shiva'

regno: 3513105

add: []

city: []

mark: [88 80 76 90 78 81 99]

Sets the value to the field ‘regno’ corresponding to student(2)

Passing Structures To Functions

An entire structure can be passed to a function, or individual fields can be passed. For example, here are two different versions of a function that calculates the profit on a software package. The profit is defined as the price minus the cost. In the first version, the entire structure variable is passed to the function, so the function must use the dot operator to refer to the *price* and *cost* fields of the input argument.

calcprof.m
function profit = calcprof(packstruct)
% Calculates the profit for a software package
% The entire structure is passed to the function
profit = packstruct.price – packstruct.cost;

>> *calcprof(package)*

ans =
19.9600

In the second version, just the *price* and *cost* fields are passed to the function using the dot operator in the function call. These are passed to two scalar input arguments in the function header, so there is no reference to a structure variable in the function itself, and the dot operator is not needed in the function.

calcprof2.m
function profit = calcprof2(oneprice, onecost)
% Calculates the profit for a software package
% The individual fields are passed to the function
profit = oneprice – onecost;
>> *calcprof2(package.price, package.cost)*

ans = 19.9600
It is important, as always with functions, to make sure that the arguments in the function call correspond one-to-one with the input arguments in the function header. In the case of *calcprof*, a structure variable is passed to an input argument, which is a structure. For the second function *calcprof2*, two individual fields, which are double values, are passed to two double arguments.

Related Structure Functions

There are several functions that can be used with structures in MATLAB. The function **isstruct** will return 1 for logical true if the variable argument is a structure variable, or 0 if not. The **isfield** function returns logical true if a fieldname (as a string) is a field in the structure argument, or logical false if not
>> *isstruct(package)*

ans = 1

The **fieldnames** function will return the names of the fields that are contained in a structure variable.
>> pack_fields = fieldnames(package)

pack_fields =
_item_no‘
_cost‘
_price‘
_code‘
Since the names of the fields are of varying lengths, the **fieldnames** function returns a cell array with the names of the fields. Curly braces are used to refer to the elements, since pack_fields is a cell array. For example, we can refer to the length of one of the strings:

>> length(pack_fields{2})

ans = 4

File I/O operations

fclose	Close one or all open files
feof	Test for end of file
ferror	File I/O error information
fgetl	Read line from file, removing newline characters
fgets	Read line from file, keeping newline characters
fileread	Read contents of file as text
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
fread	Read data from binary file
frewind	Move file position indicator to beginning of open file
fscanf	Read data from text file
fseek	Move to specified position in file

ftell	Current position
fwrite	Write data to binary file

Saving Variables To Files & Loading Variables From Files

save filename x y -ASCII

filename is the name of the file that you want to write data to.

• x, y are variables to be written to the file.

‣ If omitted, all variables are written.

• -ASCII tells Matlab to write the data in a format that you can read.

‣ If omitted, data will be written in binary format.

‣ best for large amounts of data

load filename x y

• This is the complimentary command to save.

• Reads variables x and y from file filename

‣ If variables are omitted, all variables are loaded...

FORMATTED OUTPUT IN MATLAB

disp(x) - prints the contents of variable x.

fprintf(...) - use for formatted printing

• Allows much more control over output

• Syntax: fprintf('text & formatting',variables);

• Text formatting: ‣ %a.bc

‣ a - minimum width of output buffer

‣ b - number of digits past decimal point

‣ c - formatting scheme

‣ f - floating point (typical format) 12.345

‣ e - scientific notation - 1.2345e1

‣ s - string format

```
x = [1.1 2.2 3.3 4.4];
y = 2*x;
fprintf('Hello. (%1.3f,%1.3f), (%1.1f,%1.0f)\n',...
    x(1),y(1),x(3),y(3));
```

```
Hello. (1.100,2.200), (3.3,7)
```

FILE OUTPUT IN MATLAB

• Open the file

‣ fid = fopen(filename,'w');

‣ 'w' tells matlab that we want to WRITE to the file.

‣ see "help fopen" for more information.

• Write to the file

‣ fprintf(fid,format,variables);

• Close the file

‣ fclose(fid);

FILE INPUT IN MATLAB

Import wizard "File→Import Data"

• Allows you to import data from delimited files (spreadsheets, etc)

Importing "spreadsheet" data

- • dlmread - import data from a delimited file (you choose the delimiter)
- • xlsread - import data from Excel.

General file input - three steps:

• fid=fopen(filename,'r') - open a file to allow detailed input control.

- ‣ 'r' tells matlab that we want to READ from the file.

• a=fscanf(fid,format,size);

- ‣ Works like file writing, but use fscanf rather than fprintf.
- ‣ fid - file id that you want to read from
- ‣ format - how you want to save the information (string, number)
- ‣ '%s' to read a string, '%f' to read a floating point number, '%e' to read scientific notation.

‣ size - how many entries to read.

‣ feof(fid) - returns true if end of file, false otherwise.

CREATING STRING VARIABLES

A string consists of any number of characters (including, possibly, none). These are examples of strings: _‘
_x‘
_cat‘
_Hello there‘
_123‘

A ***substring*** is a subset or part of a string. For example, _there‘ is a substring within the string _Hello there‘. ***Characters*** include letters of the alphabet, digits, punctuation marks, white space, and control characters. ***Control characters*** are characters that cannot be printed, but accomplish a task (such as a backspace or tab). ***Whitespace characters*** include the space, tab, newline (which moves the cursor down to the next line), and carriage return (which moves the cursor to the beginning of the current line). ***Leading blanks*** are blank spaces at the beginning of a string, for example, _ hello‘, and ***trailing blanks*** are blank spaces at the end of a string. There are several ways that string variables can be created. One is using assignment statements:

>> word = ‘cat’;

Another method is to read into a string variable. Recall that to read into a string variable using the **input**
function, the second argument _s‘ must be included:

>> strvar = input(‘Enter a string: ’, ‘s’)

Enter a string:
xyzabc strvar =
xyzabc

If leading or trailing blanks are typed by the user, these will be stored in the string. For example, in the following the user entered four blanks and then _xyz‘:

>> s = input(‘Enter a string: ’,‘s’)

Enter a string:
xyz s =
xyz

Strings as Vectors

Strings are treated as ***vectors of characters***—or in other words, a vector in which every element is a single character—so many vector operations can be performed. For example, the number of characters in a string can be found using the **length** function:

```
ans =
3
>> length( ‘)
ans =
1
>>
length(‘)
ans =
0
```

Notice that there is a difference between an ***empty string***, which has a length of zero, and a string consisting of a blank space, which has a length of one. Expressions can refer to an individual element (a character within the string), or a subset of a string or a transpose of a string:

```
>> mystr = ‘Hi’;

>> mystr(1)

ans
= H
>> mystr’

ans
= H
i
>> sent = ‘Hello there’;

>> length(sent)

ans
= 11
>> sent(4:8)

ans
= lo
th
```

Notice that the blank space in the string is a valid character within the string. A matrix can be created, which consists of strings in each row. So, essentially it is created as a column vector of strings, but the end result is that this would be treated as a matrix in which every element is a character:

```
>> wordmat = [‘Hello’;‘Howdy’]

wordmat
```

>> size(wordmat)

ans =
2 5

This created a 2 5 matrix of characters. With a character matrix, we can refer to an individual element, which is a character, or an individual row, which is one of the strings:

>> wordmat(2,4)

ans
= d
>> wordmat(1,:)

ans
=
Hell
o

Since rows within a matrix must always be the same length, the shorter strings must be padded with blanks so that all strings have the same length, otherwise an error will occur.

>> greetmat = ['Hello'; 'Goodbye']

??? Error using ==> vertcat

CAT arguments dimensions are not consistent.
>> greetmat = ['Hello '; 'Goodbye']

greetmat
= Hello
Goodbye

>> size(greetmat)

ans =
2 7

:

>> first = 'Bird';

>> last = 'house';

>> [first
last] ans =
Birdhouse

The function **strcat** does this also horizontally, meaning that it creates one longer string from the inputs.
>> first = 'Bird';

>> last = 'house';

>> strcat(first,last)

ans =
Birdhous
e

There is a difference between these two methods of concatenating, however, if there are leading or trailing blanks in the strings. The method of using the square brackets will concatenate the strings, including all leading and trailing blanks.

>> str1 = 'xxx ';

>> str2 = ' yyy';

>> [str1 str2]

ans =
xxx
yyy
>> length(ans)

ans
= 12

The **strcat** function, however, will remove trailing blanks (but not leading blanks) from strings before concatenating. Notice that in these examples, the trailing blanks from *str1* are removed, but the leading blanks from *str2* are not:

>> strcat(str1,str2)

ans =

length(ans)
ans =
9
>> strcat(str2,str1)

ans =
yyyxx
x
>> length(ans)

ans
= 9
The function **strvcat** will concatenate vertically, meaning that it will create a column vector of strings.
>> strvcat(first,last)

ans =
Bird
hous
e
>> size(ans)

ans =
2 5
Note that **strvcat** will pad with extra blanks automatically, in this case to make both strings have a length of 5.

Creating Customized Strings
There are several built-in functions that create customized strings, including **char, blanks**, and **sprintf**. We have seen already that the **char** function can be used to convert from an ASCII code to a character, for example:

>> char(97)

ans
= a

The **char** function can also be used to create a matrix of characters. When using the **char** function to create a matrix, it will automatically pad the strings within the rows with blanks as necessary so that they are all the same length, just like **strvcat**.

>> clear greetmat

= Hello
Goodbye
>> size(greetmat)

ans
= 2 7
The **blanks** function will create a string consisting of n blank characters which are kind of hard to see here! However, in MATLAB if the mouse is moved to highlight the result in *ans*, the blanks can be seen.

>> blanks(4)

ans =
>> length(ans)

ans
= 4

Usually this function is most useful when concatenating strings, and you want a number of blank spaces in between. For example, this will insert five blank spaces in between the words:

>> [first blanks(5) last]

ans =
Bird house

Displaying the transpose of the **blanks** function can also be used to move the cursor down. In the Command Window, it would look like this:

>> disp(blanks(4)')

This is useful in a script or function to create space in output, and is essentially equivalent to printing the newline character four times. The **sprintf** function works exactly like the **fprintf** function, but instead of printing it creates a string. Here are several examples in which the output is not suppressed so the value of the string variable is shown:

>> sent1 = sprintf('The value of pi is %.2f', pi)

sent1 =
The value of pi is 3.14
>> sent2 = sprintf('Some numbers: %5d, %2d', 33, 6)

sent2 =
Some numbers: 33, 6

= 23

In the following example, on the other hand, the output of the assignment is suppressed so the string is created including a random integer and stored in the string variable. Then, some exclamation points are concatenated to that string.

```
>> phrase = sprintf('A random integer is
%d', ›. randint(1,1,[5,10]));

>> strcat(phrase, '!!!')

ans =
A random integer is 7!!!
```

All the conversion specifiers that can be used in the **fprintf** function can also be used in the **sprintf** function.

Removing Whitespace Characters

MATLAB has functions that will remove trailing blanks from the end of a string and/or leading blanks from the beginning of a string. The **deblank** function will remove blank spaces from the end of a string. For example, if some strings are padded in a string matrix so that all are the same length, it is frequently preferred to then remove those extra blank spaces in order to actually use the string.

```
>> names = char('Sue', 'Cathy','Xavier')

names
=  Sue
Cathy
Xavier
>> name1 = names(1,:)

name1
= Sue
>> length(name1)

ans
= 6
>> name1 = deblank(name1);

>> length(name1)

ans
= 3
```

Note: The **deblank** function removes only trailing blanks from a string, not leading blanks. The

the two blanks in the middle. Selecting the result in MATLAB with the mouse would show the blank spaces.

>> strvar = [blanks(3) 'xx' blanks(2) 'yy' blanks(4)]

strvar
= xx
yy
>> length(strvar)

ans
= 13
>> strtrim(strvar)

ans
= xx
yy
>> length(ans)

ans
= 6

Changing Case

MATLAB has two functions that convert strings to all uppercase letters, or all lowercase, called **upper** and
lower.
>> mystring = 'AbCDEfgh';

>> lower(mystring)

ans =
abcdefg
h

>>
upper(ans)
ans =
ABCDEFG
H

Comparing Strings

There are several functions that compare strings and return logical true if they are equivalent, or logical false if not. The function **strcmp** compares strings, character by character. It returns

```
>> word1 = 'cat';

>> word2 = 'car';

>> word3 = 'cathedral';

>> word4 = 'CAR';

>> strcmp(word1,word2)

ans
= 0
>> strcmp(word1,word3)

ans
= 0
>> strcmp(word1,word1)

ans
= 1
>> strcmp(word2,word4)

ans
= 0
```

The function **strncmp** compares only the first *n* characters in strings and ignores the rest. The first two arguments are the strings to compare, and the third argument is the number of characters to compare (the value of *n*).

```
>> strncmp(word1,word3,3)

ans
= 1
>> strncmp(word1,word3,4)

ans
= 0
```

There is also a function **strncmpi** that compares *n* characters, ignoring the case.

Finding, Replacing, and Separating Strings

There are several functions that find and replace strings, or parts of strings, within other strings and functions that separate strings into substrings. The function **findstr** receives two strings as input arguments. It finds all occurrences of the shorter string within the longer, and returns the subscripts of the beginning of the occurrences. The order of the strings does not matter with **findstr**; it will always find the shorter string within the longer, whichever that is. The shorter

>> findstr('abcde', 'd')

ans
= 4
>> findstr('d','abcde')

ans
= 4
>> findstr('abcde', 'bc')

ans
= 2
>> findstr('abcdeabcdedd', 'd')

ans =
4 9 11 12
The function **strfind** does essentially the same thing, except that the order of the arguments does make a difference. The general form is **strfind(string, substring)**; it finds all occurrences of the substring within the string, and returns the subscripts.

>> strfind('abcdeabcde','e')

ans =
5 10
For both **strfind** and **findstr**, if there are no occurrences, the empty vector is returned.
>> strfind('abcdeabcde','ef')

ans
= [ ]
The function **strrep** finds all occurrences of a substring within a string, and replaces them with a new substring. The order of the arguments matters. The format is: strrep(string, oldsubstring, newsubstring) The following example replaces all occurrences of the substring _e' with the substring _x':

>> strrep('abcdeabcde','e','x')

ans =
abcdxabcd
x

All strings can be any length, and the lengths of the old and new substrings do not have to be the same. In addition to the string functions that find and replace, there is a function that separates a string into two substrings. The **strtok** function breaks a string into pieces; it can be called several

string, up to (but not including) the first delimiter. It also returns the rest of the string, which includes the delimiter. Assigning the returned values to a vector of two variables will capture both of these. The format is

[token rest] = strtok(string)
where *token* and *rest* are variable names. For example,

>> sentence1 = 'Hello there'

sentence1
= Hello
there
>> [word rest] = strtok(sentence1)

word
=
Hello
rest =
there
>> length(word)

ans
= 5
>> length(rest)

ans
= 6
Notice that the rest of the string includes the blank space delimiter. By default, the delimiter for the token is a whitespace character (meaning that the token is defined as everything up to the blank space), but alternate delimiters can be defined. The format

[token rest] = strtok(string, delimeters)

returns a token that is the beginning of the string, up to the first character contained within the delimiters string, and also the rest of the string. In the following example, the delimiter is the character ‗l'.

>> [word rest] = strtok(sentence1,'l')

word
= He
rest =
llo there
Leading delimiter characters are ignored, whether it is the default whitespace or a specified

firstpart
=
material
s
lastpart
=
science

Evaluating a String

The function **eval** is used to evaluate a string as a function.For example, in the following, the string _plot(x)‘ is interpreted to be a call to the **plot** function, and it produces the plot shown in Figure 6.2.

>> x = [2 6 8 3];

>> eval(‘plot(x)’)

This would be useful if the user entered the name of the type of plot to use. In this example, the string that the user enters (in this case _bar‘) is concatenated with the string _(x)‘ to create the string _bar(x)‘; this is then evaluated as a call to the **bar** function as seen in Figure 6.3. The name of the plot type is also used in the title.

The is functions for strings

There are several **is** functions for strings, which return logical true or false. The function **isletter** returns logical true if the character is a letter of the alphabet. The function **isspace** returns logical true if the character is a whitespace character. If strings are passed to these functions, they will return logical true or false for every element, or, in other words, every character.

>> isletter(‘a’)

ans
= 1
>> isletter(‘EK127’)

ans =
1 1 0 0 0
>> isspace(‘a b’)

ans
= 0 1
0

The **ischar** function will return logical true if an array is a character array, or logical false if not.

ans
= 1
>> vec = 3:5;

>> ischar(vec)

ans
= 0

Converting between string and number types

MATLAB has several functions that convert numbers to strings in which each character element is a separate digit, and vice versa. (Note: these are different from the functions **char, double**, etc., that convert characters to ASCII equivalents and vice versa.) To convert numbers to strings, MATLAB has the functions **int2str** for integers and **num2str** for real numbers (which also works with integers). The function **int2str** would convert, for example, the integer 4 to the string ‗4'.

>> rani = randint(1,1,50)

rani
= 38
>> s1 = int2str(rani)

s1
=
38
>> length(rani)

ans
= 1
>> length(s1)

ans
= 2

The variable *rani* is a scalar that stores one number, whereas *s1* is a string that stores two characters, ‗3' and ‗8'. Even though the result of the first two assignments is 38, notice that the indentation in the Command Window is different for the number and the string. The **num2str** function, which converts real numbers, can be called in several ways. If only the real number is passed to the **num2str** function, it will create a string that has four decimal places, which is the default in MATLAB for displaying real numbers. The precision can also be specified (which is the number of digits), and format strings can also be passed, as shown:

>> str2 = num2str(3.456789)

>> length(str2)

ans
= 6
>> str3 = num2str(3.456789,3)

str3
=
3.46
>> str = num2str(3.456789,'%6.2f')

str
=
3.4
6
Note that in the last example, MATLAB removed the leading blanks from the string. The function **str2num**
does the reverse; it takes a string in which a number is stored and converts it to the type **double**:

>> num = str2num('123.456')

num =
123.4560
If there is a string in which there are numbers separated by blanks, the **str2num** function will convert this to a vector of numbers (of the default type double). For example,

>> mystr = '66 2 111';

>> numvec = str2num(mystr)

numvec
= 66 2
111
>> sum(numvec)

ans
=
179

Input and Output
The previous script would be much more useful if it were more general; for example, if the value of the radius could be read from an external source rather than being assigned in the script. Also, it would be better to have the script print the output in a nice, informative way. Statements that accomplish these tasks are called ***input/output*** statements, or ***I/O*** for short. Although for

Input Function

Input statements read in values from the default or ***standard input device***. In most systems, the default input device is the keyboard, so the input statement reads in values that have been entered by the ***user***, or the person who is running the script. In order to let the user know what he or she is supposed to enter, the script must first ***prompt*** the user for the specified values. The simplest input function in MATLAB is called **input**. The **input** function is used in an assignment statement. To call it, a string is passed, which is the prompt that will appear on the screen, and whatever the user types will be stored in the variable named on the left of the assignment statement. To make it easier to read the prompt, put a colon and then a space after the prompt. For example,

>> rad = input('Enter the radius: ')

Enter the radius: 5
rad =
5

If character or string input is desired, _s' must be added after the prompt:

>> letter = input('Enter a char: ','s')

Enter a char: *g*
letter
= g

Notice that although this is a string variable, the quotes are not shown. However, they are shown in the Workspace Window. If the user enters only spaces or tabs before pressing the Enter key, they are ignored and an ***empty string*** is stored in the variable:

>> mychar = input('Enter a character: ', 's')

Enter a
character:
mychar =
''

Notice that in this case the quotes are shown, to demonstrate that there is nothing inside of the string. However, if blank spaces are entered before other characters, they are included in the string. In this example, the user pressed the space bar four times before entering -go|:

>> mystr = input('Enter a string: ', 's')

Enter a string: go
mystr =
go

>> length(mystr)

ans

>> name = input('Enter your name: ');

Enter your name: *'Stormy'* However, it is better to signify that character input is desired in the **input** function itself. Normally, the results from **input** statements are suppressed with a semicolon at the end of the assignment statements, as shown here. Notice what happens if string input has not been specified, but the user enters a letter rather than a number:

>> num = input('Enter a number: ')

Enter a number: t
??? Error using ==> input
Undefined function or
variable _t'.
Enter a number: 3

num
= 3
MATLAB gave an ***error message*** and repeated the prompt. However, if *t* is the name of a variable, MATLAB will take its value as the input:

>> t = 11;
>> num = input('Enter a number: ')

Enter a number: t
num =
11
Separate **input** statements are necessary if more than one input is desired. For example

>> x = input('Enter the x coordinate: ');

>> y = input('Enter the y coordinate: ');

Output Statements: disp **and** fprintf

Output statements display strings and the results of expressions, and can allow for ***formatting***, or customizing how they are displayed. The simplest output function in MATLAB is **disp**, which is used to display the result of an expression or a string without assigning any value to the default variable *ans*. However, **disp** does not allow formatting. For example,

>> disp('Hello')

Hello
>> disp(4^3)

64

Formatted output can be printed to the screen using the **fprintf** function. For example,

>> fprintf('The value is %d, for sure!\n',4^3)

The value is 64, for sure!

To the **fprintf** function, first a string (called the ***format string***) is passed, which contains any text to be printed as well as formatting information for the expressions to be printed. In this example, the %d is an example of format information. The %d is sometimes called a ***placeholder***; it specifies where the value of the expression that is after the string is to be printed. The character in the placeholder is called the ***conversion character***, and it specifies the type of value that is being printed. There are others, but what follows is a list of the simple placeholders:

%d integers (it actually stands for decimal integer)
%f floats
%c single characters
%s strings

Don't confuse the % in the placeholder with the symbol used to designate a comment. The character ¸\n' at the end of the string is a special character called the ***newline*** character; when it is printed the output moves down to the next line. A ***field width*** can also be included in the placeholder in **fprintf**, which specifies how many characters total are to be used in printing. For example, %5d would indicate a field width of 5 for printing an integer and %10â•›s would indicate a field width of 10 for a string. For floats, the number of decimal places can also be specified; for example, %6.2f means a field width of 6 (including the decimal point and the decimal places) with two decimal places. For floats, just the number of decimal places can also be specified; for example, %.3f if indicates three decimal places.

>> fprintf('The int is %3â•›d and the float is %6.2f\n',5,4.9)

The int is 5 and the float is 4.90 Note that if the field width is wider than necessary, ***leading*** blanks are printed, and if more decimal places are specified than necessary, ***trailing*** zeros are printed. There are many other options for the format string. For

example, the value being printed can be left-justified within the field width using a minus sign. The following example shows the difference between printing the integer 3 using %5d and using %–5d. The x‘s are just used to show the spacing.

>> fprintf(‘The integer is xx%5dxx and xx%-5dxx\n’,3,3)

The integer is xx 3xx and xx3 xx

Also, strings can be truncated by specifying decimal places:
>> fprintf(‘The string is %s or
%.4s\n’, ‘truncate’,... ‘truncate’)

The string is truncate or trun. There are several special characters that can be printed in the format string in addition to the newline character. To print a slash, two slashes in a row are used, and also to print a single quote two single quotes in a row are used. Additionally, \t is the tab character.

>> fprintf(‘Try this out: tab\t quote ‘‘ slash \\ \n’)

Try this out: tab quote _ slash \

Scripts with Input and Output

Putting all this together, we can implement the algorithm from the beginning of this chapter. The following script calculates and prints the area of a circle. It first prompts the user for a radius, reads in the radius, and then calculates and prints the area of the circle based on this radius.

script2.m
% This script calculates the area of a circle
% It prompts the user for the radius
% Prompt the user for the radius and calculate
% the area based on that radius
radius = input(_Please enter
the radius: ‘); area = pi *
(radius^2);
% Print all variables in a sentence
format fprintf(_For a circle with a
radius of %.2f,‘,radius) fprintf(_the
area is %.2f\n‘,area)

Executing the script produces the following output:

>> script2

Please enter the radius: 3.9
For a circle with a radius of 3.90, the area is 47.78

Notice that the output from the first two assignment statements is suppressed by putting semicolons at the end. That is frequently done in scripts, so that the exact format of what is displayed by the program is controlled by the **fprintf** functions.

Introduction to File Input/Output (Load **and** Sav e**)**
In many cases, input to a script will come from a data file that has been created by another source. Also, it is useful to be able to store output in an external file that can be manipulated and/or printed later. In this section, we will demonstrate how to read from an external data file, and also how to write to an external data file. There are basically three different operations, or ***modes***, on files. Files can be:

- Read from
- Written to
- A*ppended* to

Writing to a file means writing to a file, from the beginning. Appending to a file is also writing, but starting at the end of the file rather than the beginning. In other words, appending to a file means adding to what was already there. There are many different file types, which use different filename extensions. For now, we will keep it simple and just work with .dat or .txt files when working with data or text files. There are several methods for reading from files and writing to files; for now we will use the **load** function to read and the **save** function to write to files.

Writing Data to a File

The **save** function can be used to write data from a matrix to a data file, or to append to a data file. The format is:

save filename matrixvariablename –ascii.

The -ascii qualifier is used when creating a text or data file. The following creates a matrix and then saves the values of the matrix variable to a data file called testfile.dat:

>> mymat = rand(2,3)

mymat =

0.4565 0.821 0.6154
4
0.0185 0.444 0.7919
7

>> save testfile.dat mymat –ascii

This creates a file called testfile.dat that stores the numbers

0.4565 0.821 0.6154
4
0.0185 0.444 0.7919
7

The **type** command can be used to display the contents of the file; notice that scientific notation is used:

>> type testfile.dat

4.5646767e–001 8.2140716e–001 *6.1543235e–001*
1.8503643e–002 4.4470336e–001 7.9193704e–001

Note: If the file already exists, the **save** function will overwrite it; **save** always begins writing from the beginning of a file.

Appending Data to a Data File

Once a text file exists, data can be appended to it. The format is the same as previously, with the addition of the qualifier -append. For example, the following creates a new random matrix and appends it to the file just created:

>> mymat = rand(3,3)

mymat =
0.9218 0.4057 0.4103
0.7382 0.9355 0.8936
0.1763 0.9169 0.0579

>> save testfile.dat mymat –ascii –append

This results in the file testfile.dat containing

0.4565 0.821 0.6154
4
0.0185 0.444 0.7919
7
0.9218 0.405 0.4103
7

0.7382 0.9355 0.8936
0.1763 0.9169 0.0579

Note: Although technically any size matrix could be appended to this data file, in order to be able to read it back into a matrix later there would have to be the same number of values on every row.

Reading from a File

Once a file has been created (as previously), it can be read into a matrix variable. If the file is a data file, the **load** function will read from the file filename.ext (e.g., the extension might be .dat) and create a matrix with the same name as the file. For example, if the data file testfile.dat had been created as shown in the previous section, this would read from it:

>> clear

>> load testfile.dat

>> who

Your variables are:
testfile
>> testfile

testfile =
0.4565 0.8214 0.6154
0.0185 0.4447 0.7919
0.9218 0.4057 0.4103
0.7382 0.9355 0.8936
0.1763 0.9169 0.0579

Note: The **load** command works only if there are the same number of values in each line, so that the data can be stored in a matrix, and the **save** command only writes from a matrix to a file. If this is not the case, lower-level file I/O functions must be used

OPERATORS

- **Arithmetic Operations**
  Addition, subtraction, multiplication, division, power, rounding
- **Relational Operations**
  Value comparisons
- **Logical Operations**
  True or false (Boolean) conditions
- **Set Operations**
  Unions, intersection, set membership
- **Bit-Wise Operations**
  Set, shift, or compare specific bit fields

ARITHMETIC OPERATIONS

Operator	Purpose	Description
+	Addition	A+B adds A and B.
+	Unary plus	+A returns A.
-	Subtraction	A-B subtracts B from A
-	Unary minus	-A negates the elements of A.
*	Matrix multiplication	C = A*B is the linear algebraic product of the matrices A and B. The number of columns of A must equal the number of rows of B.
\	Matrix left division	x = A\B is the solution to the equation Ax = B. Matrices A and B must have the same number of rows.
/	Matrix right division	x = B/A is the solution to the equation xA = B. Matrices A and B must have the same number of columns. In terms of the left division operator, B/A = (A'\B')'.
^	Matrix power	A^B is A to the power B, if B is a scalar. For other values of B, the calculation involves eigen values and eigenvectors.
'	Complex conjugate transpose	A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.
.^	Element-wise power	A.^B is the matrix with elements A(i,j) to the B(i,j) power.
./	Right array division	A./B is the matrix with elements A(i,j)/B(i,j).
.\	Left array division	A.\B is the matrix with elements B(i,j)/A(i,j).

.'	Array transpose	A.' is the array transpose of A. For complex matrices, this does not involve conjugation.

RELATIONAL OPERATORS Conditions in **if** statements use expressions that are conceptually, or logically, either true or false. These expressions are called ***relational expressions***, or sometimes ***Boolean*** or ***logical*** expressions. These expressions can use both ***relational operators***, which relate two expressions of compatible types, and ***logical operators***, which operate on logical operands.

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

All concepts should be familiar, although the operators used may be different from those used in other programming languages, or in mathematics classes. In particular, it is important to note that the operator for equality is two consecutive equal signs, not a single equal sign (recall that the single equal sign is the assignment operator). For numerical operands, the use of these operators is straightforward.
For example,
3 < 5 means -3 less than 5,

which is conceptually a true expression. However, in MATLAB, as in many programming languages, ***logical true*** is represented by the integer 1, and ***logical false*** is represented by the integer 0. So, the expression

3 < 5 actually has the value 1 in MATLAB.
Displaying the result of expressions like this in the Command Window demonstrates the values of the expressions.

>> 3 < 5

ans = 1

>> 9 < 2

ans = 0

However, in the Workspace Window, the value shown for the result of these expressions would be true or false. The type of the result is **logical**.

Mathematical operations could be performed on the resulting 1 or 0.

>> ans + 3

ans = 4

Comparing characters, for example _a‘ < _c‘, is also possible. Characters are compared using their ASCII equivalent values. So, _a‘ < _câ•›‘ is conceptually a true expression, because the character _a‘ comes before the character _c‘.

>> ‘a’ < ‘c’

ans = 1

LOGICAL OPERATORS

Symbol	Role
&	Logical AND
|	Logical OR
&&	Logical AND (with short-circuiting)
||	Logical OR (with short-circuiting)
~	Logical NOT

All logical operators operate on logical or Boolean operands. The **not** operator is a unary operator; the others are binary. The **not** operator will take a Boolean expression, which is conceptually true or false, and give the opposite value. For example, (3 < 5) is conceptually false since (3 < 5) is true. The **or** operator has two Boolean expressions as operands. The result is true if either or both of the operands are true, and false only if both operands are false. The **and** operator also operates on two Boolean operands. The result of an **and** expression is true only if both operands are true; it is false if either or both are false. In addition to these logical operators, MATLAB also has a function **xor**, which is the exclusive or function. It returns logical true if one (and only one) of the arguments is true. For example, in the following only the first argument is true,

so the result is true:
>> xor(3 < 5, ‘a’ > ‘c’)

ans = 1In this example, both arguments are true so the result is false:
>> xor(3 < 5, ‘a’ < ‘c’)

ans = 0

Given the logical values of true and false in variables x and y, the ***truth table*** shows how the logical operators work for all combinations. Note that the logical operators are commutative

SPECIAL CHARACTERS

Symbol	Role
,	Use commas to separate row elements in an array, array subscripts, function input and output arguments, and commands entered on the same line.
:	Use the colon operator to create regularly spaced vectors, index into arrays, and define the bounds of a for loop.
;	Use semicolons to separate rows in an array creation command, or to suppress the output display of a line of code.
( )	Use parentheses to specify precedence of operations, enclose function input arguments, and index into an array
[]	Square brackets enable array construction and concatenation, creation of empty matrices, deletion of array elements, and capturing values returned by a function.
{}	Use curly braces to construct a cell array, or to access the contents of a particular cell in a cell array.
%	The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code.
‘ ’	Use single quotes to create character vectors that have class char

OPERATOR PRECEDENCE RULES

1. Parentheses ()
2. Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)
3. Power with unary minus (.^-), unary plus (.^+), or logical negation (.^~) as well as matrix power with unary minus (^-), unary plus (^+), or logical negation (^~).
4. Unary plus (+), unary minus (-), logical negation (~)
5. Multiplication (.*), right division (./), left division (.\), matrix multiplication (*), matrix right division (/), matrix left division (\)
6. Addition (+), subtraction (-)
7. Colon operator (:)
8. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
9. Element-wise AND (&)
10. Element-wise OR (|)
11. Short-circuit AND (&&)
12. Short-circuit OR (||)

BRANCHES & LOOPS

BRANCHES are MATLAB statements that permit us to select and execute specific section of code called blocks, while skipping other sections of code.

- if
- switch
- try-catch

LOOPS are MATLAB construct that allow us to execute a sequence of statements more than once

- for
- while

The If **Statement**

The **if** statement chooses whether or not another statement, or group of statements, is executed. The general form of the **if** statement is:

```
if condition
        action
end
```

A ***condition*** is a relational expression that is conceptually, or logically, either true or false. The ***action*** is a statement, or a group of statements, that will be executed if the condition is true. When the **if** statement is executed, first the condition is evaluated. If the value of the condition is conceptually true, the action will be executed, and if not, the action will not be executed. The action can be any number of statements until the reserved word **end**; the action is naturally bracketed by the reserved words **if** and **end**. (Note: This is different from the **end** that is used as an index into a vector or matrix.)

For example, the following **if** statement checks to see whether the value of a variable is negative. If it is, the value is changed to a positive number by using the absolute value function; otherwise nothing is changed.

```
if num < 0
        num = abs(num)
end
```

If statements can be entered in the Command Window, although they generally make more sense in scripts or functions. In the Command Window, the **if** line would be entered, then the Enter key, then the action, the Enter key, and finally **end** and Enter; the results will immediately follow. For example, the previous **if** statement is shown twice here. Notice that the output from the assignment is not suppressed, so the result of theÂ€action will be shown if the action is executed. The first time the value of the variable is negative so the action is executed and the variable is modified, but in the second case the variable is positive so the action is skipped.

```
>> num = −4;

>> if num < 0
```

```
        num = abs(num)
        end
        num = 4

>> num = 5;
>> if num < 0
        num = abs(num)
end
>>
```

This may be used, for example, to make sure that the square root function is not used on a negative number. The following script prompts the user for a number, and prints the square root. If the user enters a negative number, the **if** statement changes it to positive before taking the square root.

```
sqrtifexamp.m
% Prompt the user for a number and print its
sqrt num = input('Please enter a number: ');
% If the user entered a negative number,
change it
if num < 0
num = abs(num);
 end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

Here are two examples of running this script:

```
>> sqrtifexamp

Please enter a number: −4.2
The sqrt of 4.2 is 2.0
>> sqrtifexamp

Please enter a number: 1.44
 The sqrt of 1.4 is 1.2
```

In this case, the action of the **if** statement was a single assignment statement. The action can be any number of valid statements. For example, we may wish to print a note to the user to say that the number entered was being changed.

```
sqrtifexampii.m
% Prompt the user for a number and print its sqrt num =
input('Please enter a number: ');
```

```
% If the user entered a negative number, tell
% the user and change it
 if num < 0
disp('OK, we''ll use the absolute value')
 num = abs(num);
end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

>> *sqrtifexampii*

Please enter a number: −25 OK, we‘ll use the
absolute value
The sqrt of 25.0 is 5.0
Notice the use of two single quotes in the **disp** statement in order to print one single quote

The If-Else **statement**

The **if** statement chooses whether an action is executed or not. Choosing between two actions, or choosing from several actions, is accomplished using **if-else**, nested **if**, and **switch** statements. The **if-else** statement is used to choose between two statements, or sets ofstatements.

The general form is:
if condition
action1
else
action2
end

First, the condition is evaluated. If it is conceptually true, then the set of statements designated as action1 is executed, and that is it for the **if-else** statement. If instead the condition is conceptually false, the second set of statements designated as action2 is executed, and that‘s it. The first set of statements is called the action of the **if** clause; it is what will be executed if the expression is true. The second set of statements is called the action of the **else** clause; it is what will be executed if the expression is false. One of these actions, and only one, will be executed—which one depends on the value of the condition. For example, to determine and print whether or not a random number in the range from 0 to 1 is less than 0.5, an **if-else** statement could be used:

```
if rand < 0.5
disp(‘It was less than .5!‘)
else
disp(‘It was not less than .5!‘)
end
```

One application of an **if-else** statement is to check for errors in the inputs to a script. For example, an earlier script prompted the user for a radius, and then used that to calculate the area of a circle. However, it did not check to make sure that the radius was valid (e.g., a positive number). Here is a modified script that checks the radius:

checkradius.m

```
% This script calculates the area of a circle
% It error-checks the user‘s radius
radius = input(‘Please enter the radius: ‘);
if radius <= 0
        fprintf(_Sorry; %.2f is not a valid radius\n‘,radius)
else        end
```

Examples of running this script when the user enters invalid and then valid radii are shown here:
>> *checkradius*

Please enter the radius: −4
Sorry; −4.00 is not a valid radius

>> *checkradius*

Please enter the radius: 5.5
For a circle with a radius of 5.50, the area is 95.03

The **if-else** statement in this example chooses between two actions: printing an error message, or actually using the radius to calculate the area, and then printing out the result. Notice that the action of the **if** clause is a single statement, whereas the action of the **else** clause is a group of three statements.

Nested If-Else Statements

The **if-else** statement is used to choose between two statements. In order to choose from more than two statements, the **if-else** statements can be nested, one inside of another. For example, consider implementing the following continuous

mathematical function y = f(x):
y = 1 for x < −1
y = x2 for −1 ≤ x ≤ 2
y = 4 for x > 2

The value of y is based on the value of x, which could be in one of three possible ranges. Choosing which range could be accomplished with three separate **if** statements, as follows:

```
if x < −1
        y = 1;
end
if x > = −1 && x < = 2
         y = x^2;
end
if x > 2

end     y= 4;
```

Since the three possibilities are mutually exclusive, the value of *y* can be determined by using three separate **if** statements. However, this is not very efficient code: all three Boolean expressions must be evaluated, regardless of the range in which *x* falls. For example, if *x* is less than –1, the first expression is true and 1 would be assigned to *y*. However, the two expressions in the next two **if** statements are still evaluated. Instead of writing it this way, the expressions can be ***nested*** so that the statement ends when an expression is found to be true

```
% If we are here, x must be > = −1
% Use an if-else statement to choose
% between the two remaining ranges
  if x > = −1 && x < = 2
y = x^2;
else
% No need to check
% If we are here, x must be > 2
y = 4;
end
end
```

By using a nested **if-else** to choose from among the three possibilities, not all conditions must be tested as they were in the previous example. In this case, if *x* is less than –1, the statement to assign 1 to *y* is executed, and the **if-else** statement is completed so no other conditions are tested. If, however, *x* is not less than –1, then the else clause is executed. If the else clause is executed, then we already know that *x* is greater than or equal to –1 so that part does not need to be tested. Instead, there are only two remaining possibilities: either *x* is less than or equal to 2, or it is greater than 2. An **if-else** statement is used to choose between those two possibilities. So, the action of the **else** clause was another **if-else** statement. Although it is long, this is one **if-else** statement, a nested **if-else** statement. The actions are indented to show the structure. Nesting **if-else** statements in this way can be used to choose from among three, four, five, six, or more options—the possibilities are practically endless! This is actually an example of a particular kind of nested **if-else** called a cascading **if-else** statement. In this type of nested **if-else** statement, the conditions and actions cascade in a stair-like pattern.

For example, if there are *n* choices (where $n > 3$ in this example), the following general form

would be used:
```
if condition1
        Action1
        elseif condition2
        action2
        elseif condition3
        action3
% etc: there can be many of
these else
        actionn % the nth action
end
```

The actions of the **if**, **elseif**, and **else** clauses are naturally bracketed by the reserved words **if**, **elseif**, **else**, and **end**. For example, the previous example could be written using the **elseif** clause rather than nesting **if- else** statements:

So, there are three ways of accomplishing this task: using three separate **if** statements, using

of y:

Another example demonstrates choosing from more than just a few options. The following function receives an integer quiz grade, which should be in the range from 0 to 10. The program then returns a corresponding letter grade, according to the following scheme: a 9 or 10 is an ‗A', an 8 is a ‗B', a 7 is a ‗C', a 6 is a ‗D', and anything below that is an ‗F'. Since the possibilities are mutually exclusive, we could implement the grading scheme using separate **if** statements. However, it is more efficient to have one **if- else** statement with multiple **elseif** clauses. Also, the function returns the value ‗X' if the quiz grade is not valid. The function does assume that the input is an integer.

```
letgrade.m
function grade = letgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument
% First, error-check
 if quiz < 0 || quiz > 10
        grade = ‗X';
% If here, it is valid so figure out the
% corresponding letter grade
elseif quiz == 9 || quiz == 10
grade = ‗A';
elseif quiz == 8
        grade = ‗B';
 elseif quiz == 7
        grade = ‗C';
elseif quiz == 6
        grade = ‗D';
else
        grade = ‗F';
end
```

Here are three examples of calling this function:

```
>> quiz = 8;

>> lettergrade = letgrade(quiz)

lettergrade
= B
>> quiz = 4;

>> letgrade(quiz)
```

>> quiz = 22;

>> lg = letgrade(quiz)

lg = X

In the part of this **if** statement that chooses the appropriate letter grade to return, all the Boolean expressions are testing the value of the variable *quiz* to see if it is equal to several possible values, in sequence (first 9 or 10, then 8, then 7, etc.). This part can be replaced by a **switch** statement.

The Switch Statement

A **switch** statement can often be used in place of a nested **if-else** or an **if** statement with many **elseif** clauses. Switch statements are used when an expression is tested to see whether it is *equal to* one of several possible values. The general form of the **switch** statement is:

```
   switch
   switch_expression
   case caseexp1
         action1
   case caseexp2
         action2
    case caseexp3
         action3
 % etc: there can be many of
these otherwise
         actionn
end
```

The **switch** statement starts with the reserved word **switch**, and ends with the reserved word **end**. The switch_expression is compared, in sequence, to the case expressions (caseexp1, caseexp2, etc.). If the value of the switch_expression matches caseexp1, for example, then action1 is executed and the **switch** statement ends. If the value matches caseexp3, then action3 is executed, and in general if the value matches caseexpi, where i can be any integer from 1 to n, then actioni is executed. If the value of the switch_expression does not match any of the case expressions, the action after the word **otherwise** is executed. For the previous example, the **switch** statement can be used as follows:

switchletgrade.m

```
function grade = switchletgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument using switch
% First, error-check if quiz < 0 || quiz > 10
        grade = _X‘;
else
        % If here, it is valid so figure out the
        % corresponding letter grade using a
        switch switch quiz
        case 10
                grade= 'A‘;
        case 9
                grade = _D‘;
        case 8
   grade=‘B‘;
        case 7
   grade= ‘A‘;
        case 6
        otherwise
                grade = _F‘;
        end
end
```

Here are two examples of calling this function:

```
>> quiz = 22;

>> lg = switchletgrade(quiz)

lg = X
>> quiz = 9;

>> switchletgrade(quiz)

ans
= A
```

Note that it is assumed that the user will enter an integer value. If the user does not, either an error message will be printed or an incorrect result will be returned. Since the same action of printing _A‘ is desired for more than one case, these can be combined as follows:

```
switch quiz
        case {10,9}
                grade = _A‘;
        case 8
                grade = _B‘;
        % etc.
```

(The curly braces around the case expressions 10 and 9 are necessary.) In this example, we error-checked first using an **if-else** statement, and then if the grade was in the valid range, used a **switch** statement to find the corresponding letter grade.

Sometimes the **otherwise** clause is used instead for the error message. For example, if the user is supposed to enter only a 1, 3, or 5, the script might be organized as follows:

switcherror.m

```
% Example of otherwise for error message choice =
input('Enter a 1, 3, or 5: ');
switch choice
        case 1
                disp(_It''s a one!!')
        case 3
                disp(_It''s a three!!')
        case 5
                disp(_It''s a five!!')
           otherwise
                   disp(_Follow directions next time!!')
end
```

In this case, actions are taken if the user correctly enters one of the valid options. If the user does not, the **otherwise** clause handles printing an error message. Note the use of two single quotes within the string to print one.

>> switcherror

Enter a 1, 3, or 5: 4
Follow directions next time!!

The for **Loop**

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times. The variable that is used to iterate through values is called a ***loop variable***, or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although variable names in general should be mnemonic, it is common for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i, j, k, l*, etc.) This is historical, and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in values for -1 , so using either as a loop variable will override that value. If that is not an issue, then it is acceptable to use *i* as a loop variable. The general form of the **for** loop is:

```
for loopvar = range
        action
end
```

where *loopvar* is the loop variable, *range* is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**. The range can be specified using any vector, but normally the easiest way to specify the range of values is to use the colon operator. As an example, to print a column of numbers from 1 to 5:

```
for i = 1:5
        fprintf(_%d\n‘,i)
end
```

This loop could be entered in the Command Window, although like **if** and **switch** statements, loops will make more sense in scripts and functions. In the Command Window, the results would appear after the **for** loop:

>> for i = 1:5
fprintf(‘%d\n’,i)

end

4
5

What the **for** statement accomplished was to print the value of *i* and then the newline character for every value of *i*, from 1 through 5 in steps of 1. The first thing that happens is that *i* is initialized to have the value
1. Then, the action of the loop is executed, which is the **fprintf** statement that prints the value of *i* (1), and then the newline character to move the cursor down. Then, *i* is incremented to have the value of 2. Next, the action of the loop is executed, which prints 2 and the newline. Then, *i* is incremented to 3 and that is printed, then *i* is incremented to 4 and that is printed, and then finally *i* is incremented to 5 and that is printed. The final value of *i* is 5; this value can be used once the loop has finished.

Finding Sums and Products

A very common application of a **for** loop is to calculate sums and products. For example, instead of just printing the integers 1 through 5, we could calculate the sum of the integers 1 through 5 (or, in general, 1 through *n*, where *n* is any positive integer). Basically, we want to implement or calculate the sum 1 + 2 + 3
+ … + n. In order to do this, we need to add each value to a ***running sum***. A running sum is a sum that will keep changing; we keep adding to it. First the sum has to be initialized to 0, then in this case it will be 1 (0 + 1), then 3 (0 + 1 + 2), then 6 (0 + 1 + 2 + 3), and so forth. In a function to calculate the sum, we need a loop or iterator variable *i*, as before, and also a variable to store the running sum. In this case we will use the output argument *runsum* as the running sum. Every time through the loop, the next value of *i* is added to the value of *runsum*. This function will return the end result, which is the sum of all integers from 1 to the input argument *n* stored in the output argument *runsum*.

```
sum_1_to_n.m
function runsum = sum_1_to_n(n)
% This function returns the sum of
% integers from 1 to
n runsum = 0;
for i = 1:n
        runsum = runsum + i;
end
```

As an example, if 5 is passed to be the value of the input argument *n*, the function will calculate and return 1 + 2 + 3 + 4 + 5, or 15:

```
>> sum_1_to_n(5)

ans
= 15
```

Note that the output was suppressed when initializing the sum to 0 and when adding to it during the loop. Another very common application of a **for** loop is to find a ***running product***. For example, instead of finding the sum of the integers 1 through n, we could find the product of the integers 1 through n. Basically, we want to implement or calculate the product 1 * 2 * 3 * 4 *… * n, which is called the ***factorial*** of n, written n!.

For Loops that Do Not Use the Iterator Variable in the Action

In all the examples that we have seen so far, the value of the loop variable has been used in some way in the action of the **for** loop: we have printed the value of *i*, or added it to a sum, or multiplied it by a running product, or used it as an index into a vector. It is not always necessary to actually use the value of the loop variable, however. Sometimes the variable is simply used to iterate, or repeat, a statement a specified number of times. For example,

```
for i = 1:3
        fprintf(_I will not chew gum\n‘)
end
```

produces the output:

```
I will not chew gum
I will not chew gum
I will not chew gum
```

The variable *i* is necessary to repeat the action three times, even though the value of *i* is not used in the action of the loop.

Nested for **Loops**

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a ***nested loop***. As an example, a nested **for** loop will be demonstrated in a script that will print a box of *‘s. Variables in the script will specify how many rows and columns to print. For example, if *rows* has the value 3, and *columns* has the value 5, the

output would be:

```
*****
*****
*****
```

Since lines of output are controlled by printing the newline character, the basic

- algorithm is: For every row of output,
- Print the required number of *‘s
- Move the cursor down to the next line (print the _\n‘)

```
printstars.m
% Prints a box of stars
% How many will be specified by 2 variables
% for the number of rows and
columns rows = 3;
columns = 5;
% loop over the
rows for i=1:rows
        % for every row loop to print *‘s and then one \n for j=1:columns
fprintf(_*‘)
        end
         fprintf(_\n‘)
end
```

Running the script displays the output:

```
>> printstars

*****
*****
*****
```

The variable *rows* specifies the number of rows to print, and the variable *columns* specifies how many *‘s to print in each row. There are two loop variables: *i* is the loop variable for the rows, and *j* is the loop variable for the columns. Since the number of rows and columns are known (given by the variables *rows* and *columns*), **for** loops are used. There is one **for** loop to loop over the rows, and another to print the required number of *‘s. The values of the loop variables are not used within the loops, but are used simply to iterate the correct number of times. The first **for** loop specifies that the action will be repeated *rows* times. The action of this loop is to print *‘s and then the newline character. Specifically, the action is to loop to print *columns* *‘s across on one line. Then, the newline character is printed after all five stars to move the cursor down for the next line. The first **for** loop is called the ***outer loop***; the second **for** loop is called the ***inner loop***. So, the outer loop is over the rows, and the inner loop is over the columns. The outer loop must be over the rows because the program is printing a certain number of rows of output. For each row, a loop is necessary to print the required number of *‘s; this is the inner **for** loop. When this script is executed, first the outer loop variable *i* is initialized to 1. Then, the action is executed. The action consists of the inner loop, and then printing the newline character. So, while the outer loop variable has the value 1, the inner loop variable *j* iterates through all its values. Since the value of *columns* is 5, the inner loop will print a * five times. Then, the newline character is printed and the outer loop variable *i* is incremented to 2. The action of the outer loop is then executed again, meaning the inner loop will print five *‘s, and then the newline character will be printed. This continues, and in all, the action of the outer loop will be executed *rows* times. Notice the action of the outer loop consists of two statements (the **for** loop and an **fprintf** statement). The action of the inner loop, however, is only a single statement. The **fprintf** statement to print the newline character must be separate from the other **fprintf** statement that

prints the *. If we simply had fprintf(_*\n‘) as the action of the inner loop, this would print a long column of 15 *‘s, not a box In these examples, the loop variables were used just to specify the number of times the action is to be repeated. These same loops could be used instead to produce

a multiplication table by multiplying the values of the loop variables. The following function *multtable* calculates and returns a matrix that is a multiplication table. Two arguments are passed to the function, which are the number of rows and columns for this matrix

multtable.m

```
function outmat = multtable (rows, columns)
% Creates a matrix which is a multiplication table
% Preallocate the matrix
outmat = zeros(rows,columns);
for i = 1:rows
        for j = 1:columns
                outmat(i,j) = i * j;
        end
end
```

In the following example, the matrix has three rows and five columns:

```
>> multtable(3,5)

ans =
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
```

Notice that this is a function that returns a matrix; it does not print anything. It preallocates the matrix to zeros, and then replaces each element. Since the number of rows and columns are known, **for** loops are used. The outer loop loops over the rows, and the inner loop loops over the columns. The action of the nested loop calculates i * j for all values of *i* and *j*. First, when *i* has the value 1, *j* iterates through the values 1 through 5, so first we are calculating 1 * 1, then 1 * 2, then 1 * 3, then 1 * 4, and finally 1 * 5. These are the values in the first row (first in element (1,1), then (1,2), then (1,3), then (1,4), and finally (1,5)). Then, when *i* has the value 2, the elements in the second row of the output matrix are calculated, as *j* again iterates through the values from 1 through 5. Finally, when *i* has the value 3, the values in the third row are calculated (3 * 1, 3 * 2, 3 * 3, 3 * 4, and 3 * 5). This function could be used in a script that prompts the user for the number of rows and columns, calls this function to return a multiplication table, and writes the resulting matrix to a file:

createmulttab.m

```
% Prompt the user for rows and columns and
% create a multiplication table to store in
% a file mymulttable.dat
num_rows = input('Enter the number of rows: ');
num_cols = input('Enter the number of columns: ');
multmatrix = multtable(num_rows, num_cols);
save mymulttable.dat multmatrix –ascii
```

Here is an example of running this script, and then loading from the file into a matrix in order to verify that the file was created:

Enter the number of rows: 6
Enter the number of
columns: 4
>> load mymulttable.dat

>> mymulttable

mymulttable =

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16
5	10	15	20
6	12	18	24

Logical Vectors

The relational operators can also be used with vectors and matrices. For example, let‘s say that there is a vector, and we want to compare every element in the vector to 5 to determine whether it is greater than 5 or not. The result would be a vector (with the same length as the original) with logical true or false values. Assume a variable vec as shown here.

>> vec = [5 9 3 4 6 11];

In MATLAB, this can be accomplished automatically by simply using the relational operator >.
>> isg = vec > 5

isg =

0	1	0	0	1	1

Notice that this creates a vector consisting of all logical true or false values. Although this is a vector of ones and zeros, and numerical operations can be done on the vector *isg*, its type is **logical** rather than **double**.
>> doubres = isg + 5

ans =

5	6	5	5	6	6

>> whos

Name	Size	Bytes	Class
doubres	1x6 4	8	double array
isg	1x6	6	logical array
vec	1x6	48	double array

To determine how many of the elements in the vector *vec* were greater than 5, the **sum** function could be used on the resulting vector *isg*:

>>
sum(isg)
ans =
3

```
>> vec(isg)

ans =
9       6       11
```

Because the values in the vector must be **logical** 1‘s and 0‘s, the following function that appears at first to accomplish the same operation using the programming method, actually does not. The function receives two input arguments: the vector, and an integer with which to compare (so it is somewhat more general). It loops through every element in the input vector, and stores in the result vector either a 1 or 0 depending on whether vec(i) > n is true or false.

```
testvecgtn.m
function outvec = testvecgtn(vec,n)
% Compare each element in vec to see whether it
% is greater than n or not
% Preallocate the vector
outvec = zeros(size(vec));
for i = 1:length(vec)
        % Each element in the output vector stores 1 or 0
         if vec(i) > n
         outvec(i) = 1;
          else
         outvec(i) = 0;
         end
end
```

Calling the function appears to return the same vector as simply vec > 5, and summing the result still works to determine how many elements were greater than 5.

```
>> notlog = testvecgtn(vec,5)

notlog =
0       1       0       0       1       1
>> sum(notlog)

ans =
        3
```

However, as before, it could not be used to index into a vector because the elements are **double**, not **logical**:

```
>> vec(notlog)

??? Subscript indices must either be real positive integers or logicals.
```

While **Loops**

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action

```
while condition
        action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is true. The condition must eventually become false to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.) The way it works is that first the condition is evaluated. If it is logically true, the action is executed. So, to begin with it is just like an **if** statement. However, at that point the condition is evaluated again. If it is still true, the action is executed again. Then, the action is evaluated again. If it is still true, the action is executed again. Then, the action is… eventually, this has to stop! Eventually something in the action has to change something in the condition so it becomes false. As an example of a conditional loop, we will write a function that will find the first factorial that is greater than the input argument *high*. Previously, we wrote a function to calculate a particular factorial. For example, to calculate 5! we found the product 1 * 2 * 3 * 4 * 5. In that case a **for** loop was used, since it was known that the loop would be repeated five times. Now, we do not know how many times the loop will be repeated. The basic algorithm is to have two variables, one that iterates through the values 1, 2, 3, and so on, and one that stores the factorial of the iterator at each step. We start with 1, and 1 factorial, which is 1. Then, we check the factorial. If it is not greater than *high*, the iterator variable will then increment to 2, and find its factorial (2). If this is not greater than *high*, the iterator will then increment to 3, and the function will find its factorial (6). This continues until we get to the first factorial that is greater than *high*. So, the process of incrementing a variable and finding its factorial is repeated until we get to the first value greater than *high*. This is implemented using a **while** loop:

factgthigh.m
```
function facgt = factgthigh(high)
% Finds the first factorial >
high i=0;
fac=1;
while fac <= high
        i=i+1;
        fac = fac * i;
end
facgt = fac;
```

Here is an example of calling the function, passing 5000 for the value of the input argument *high*.
>> factgthigh(5000)

```
ans
=
5040
```

The iterator variable *i* is initialized to 0, and the running product variable *fac*, which will store the factorial of each value of *i*, is initialized to 1. The first time the **while** loop is executed, the condition is conceptually true: 1 is less than or equal to 5000. So, the action of the loop is

executed, which is to increment *i* to 1 and *fac* to 1 (1 * 1). After the execution of the action of the loop, the condition is evaluated again. Since it will still be true, the action is executed: *i* is incremented to 2, and *fac* will get the value 2 (1 * 2). The value 2 is still <= 5000, so the action

will be executed again: *i* will be incremented to 3, and *fac* will get the value 6 (2 * 3). This continues until the first value of *fac* is found that is greater than 5000. As soon as *fac* gets to this value, the condition will be false and the **while** loop will end. At that point the factorial is assigned to the output argument, which returns the value. The reason that *i* is initialized to 0 rather than 1 is that the first time the loop action is executed, *i* becomes 1 and *fac* becomes 1 so we have 1 and 1!, which is 1. Notice that the output of all assignment statements is suppressed in the function.

Multiple Conditions in a While **Loop**

In the previous section, we wrote a function *myany* that imitated the built-in **any** function by returning logical true if any value in the input vector was logical true, and logical false otherwise. The function was inefficient because it looped through all the elements in the input vector, even though once one logical true value is found it is no longer necessary to examine any other elements. A **while** loop will improve on this. Instead of looping through all the elements, what we really want to do is to loop until either a logical true value is found, or until we've gone through the entire vector. Thus, we have two parts to the condition in the **while** loop. In the following function, we initialize the output argument to logical false, and an iterator variable *i* to
The action of the loop is to examine an element from the input vector: if it is logical true, we change the output argument to be logical true. Also in the action the iterator variable is incremented. The action of the loop is continued as long as the index has not yet reached the end of the vector, and as long as the output argument is still logical false.

myanywhile.m

```
function logresult = myanywhile(vec)
% Simulates the built-in function any
% Uses a while loop so that the action halts
% as soon as any true value is
found logresult = logical(0);
i = 1;
while i <= length(vec) && logresult == 0
        if vec(i) = 0
        logresult = logical(1);
         end
        i = i + 1;
end
```

The output produced by this function is the same as the *myany* function, but it is more efficient because now as soon as the output argument is set to logical true, the loop ends.

Debugging Techniques

Any error in a computer program is called a ***bug***. This term is thought to date back to the 1940s, when a problem with an early computer was found to have been caused by a moth in the computer‘s circuitry! The process of finding errors in a program, and correcting them, is still called ***debugging***.

Types of Errors

There are several different kinds of errors that can occur in a program, which fall into the categories of ***syntax errors, run-time errors***, and ***logical errors***. Syntax errors are mistakes in using the language. Examples of syntax errors are missing a comma or a quotation mark, or misspelling a word. MATLAB itself will flag syntax errors and give an error message. For example, the following string is missing the end quote:

>> mystr = ‘how are you;

??? mystr = ‘how are you;
|
Error: A MATLAB string constant is not terminated properly.

Another common mistake is to spell a variable name incorrectly, which MATLAB will also catch.

>> value = 5;

>> newvalue = valu + 3;

??? Undefined function or variable ‘valu’.

Run-time, or execution-time, errors are found when a script or function is executing. With most languages, an example of a run-time error would be attempting to divide by zero. However, in MATLAB, this will generate a warning message. Another example would be attempting to refer to an element in an array that does not exist.

runtime_ex.m

```
% This script shows an execution-time
error vec = 3:5;
for i = 1:4
        disp(vec(i))
end
```

This script initializes a vector with three elements, but then attempts to refer to a fourth. Running it prints the three elements in the vector, and then an error message is generated when it attempts to refer to the fourth element. Notice that it gives an explanation of the error, and it gives the line number in the script in which the error occurred.

```
>> runtime_ex

3
4
5
??? Attempted to access vec(4); index out of bounds because
numel(vec)=3.
Error in ==> runtime_ex at 6
        disp(vec(i))
```

Logical errors are more difficult to locate, because they do not result in any error message. A

logical error is a mistake in reasoning by the programmer, but it is not a mistake in the programming language. An example of a logical error would be dividing by 2.54 instead of multiplying in order to convert inches to centimeters. The results printed or returned would be incorrect, but this might not be obvious. All programs should be robust and should wherever possible anticipate potential errors, and guard against them. For example, whenever there is input into a program, the program should error-check and make sure that the input is in the correct range of values. Also, before dividing, the denominator should be checked to make sure that it is not zero. Despite the best precautions, there are bound to be errors in programs.

Tracing

Many times, when a program has loops and/or selection statements and is not running properly, it is useful in the debugging process to know exactly which statements have been executed. For example, here is a function that attempts to display In Middle Of Range if the argument passed to it is in the range from 3 to 6, and Out Of Range otherwise.

```
testifelse.m
function testifelse(x)
% This function will test the
debugger if 3 < x < 6
        disp('In middle of range')
else

   disp('Out of range')
end
```

However, it seems to print In Middle Of Range for all values of x:

```
>> testifelse(4)

In middle of range
>> testifelse(7)

In middle of range
>> testifelse(–2)
```

In middle of range

One way of following the flow of the function, or ***tracing*** it, is to use the **echo** function. The **echo** function, which is a toggle, will display every statement as it is executed as well as results from the code. For scripts, just **echo** can be typed, but for functions, the name of the function must be specified, for example, echo function name on/off
>> echo testifelse on

Editor/Debugger

MATLAB has many useful functions for debugging, and debugging can also be done through its editor, called the Editor/Debugger. Typing **help debug** at the prompt in the Command Window will show some of the debugging functions. Also, in the Help Browser, clicking the Search tab and then typing **debugging** will display basic information about the debugging processes. It can be seen in the previous example that the action of the **if** clause was executed and it printed In Middle Of Range, but just from that it cannot be determined why this happened. There are several ways to set ***breakpoints*** in a file (script or function) so that the variables or expressions can be examined. These can be done from the Editor/Debugger, or commands can be typed from the Command Window. For example, the following **dbstop** command will set a breakpoint in the fifth line of this function (which is the action of the **if** clause), which allows us to type variable names and/or expressions to examine their values at that point in the execution. The function **dbcont** can be used to continue the execution, and **dbquit** can be used to quit the debug mode. Notice that the prompt becomes K>> in debug mode.

```
>> dbstop testifelse 5

>> testifelse(–2)

5 disp(_In middle of range‘)
K>> x

x =
          –2
K>> 3 < x

ans =
          0
K>> 3 < x < 6

ans =
          1
K>> dbcont

In middle
of range
end
>>
```

By typing the expressions 3 < x and then 3 < x < 6, we can determine that the expression 3 < x will return either 0 or 1. Both 0 and 1 are less than 6, so the expression will always be true, regardless of the value of x!

Function Stubs

Another common debugging technique, which is used when there is a script main program that calls many functions, is to use ***function stubs***. A function stub is a placeholder, used so that the script will work even though that particular function hasn‘t been written yet. For example, a programmer might start with a script main program that consists of calls to three function that accomplish all the tasks.

mainmfile.m

```
% This program gets values for x and y, and
% calculates and prints
z [x, y] = getvals;
z = calcz(x,y);
printall(x,y,z)
```

The three functions have not yet been written, however, so function stubs are put in place so that the script can be executed and tested. The function stubs consist of the proper function headers, followed by a simulation of what the function will eventually do (e.g., it puts arbitrary values in for the output arguments).

getvals.m

```
function [x, y] = getvals
x = 33;
y = 11;
```

calcz.m

```
function z = calcz(x,y)
z = 2.2;
```

printall.m

```
function printall(x,y,z)
disp(‘Something‘)
```

Then, the functions can be written and debugged one at a time. It is much easier to write a working program using this method than to attempt to write everything at once—then, when errors occur, it is not always easy to determine where the problem

QUESTION BANK

PART A	CO
1. List the various windows available in MATLAB?	1
2. What is matrix?	1
3. List 5 built in functions in MATLAB	1
4. Develop MATLAB function for finding the determinant of a matrix?	1
5. List the various relational operators in the MATLAB?	1
6. What is the use of strcat functions	1
7. Compare floor and ceil functions in MATLAB	1
8. What is the use of getfield command in MATLAB?	1
9. What is the need for celldisp command in MATLAB?	1
10. What is the use of CHAR command in MATLAB	1

PART B

1. Design a string calculator in MATLAB that performs various string operations Compare and contrast the various input and output statements in MATLAB support your answer with suitable examples. 1
2. Develop a cell array(a) whose size is 2*2. 1

   a(1,1)= ‘ India Australia Matches’
   a(1,2)=[50 , 60, 100]
   a(2,1)=’Viratkohli’
   a(2,2)=[ ];

   use preallocation and assignment statements .Also display 60. 1

3.	Develop a software to assist the mentor in maintaining the address details of each student use structure array	1
4.	Explain the various ways of reading and writing data into files. Illustrate with suitable examples	1
5.	Explain the various data types used in MATLAB .support your answer with suitable examples .	1
6.	Develop a function foe finding factorial of a number. Also design MATLAB codes for finding binomial co-efficient.	1
7.	Explain different methods of string and accessing values from matrices and vectors in MA	1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT – II – SIMULINK– SEIA1503

UNIT II- SIMULINK

Introduction-Introduction to simulink libraries-graphical user interface - selection of objects- blocks- lines- simulationapplicationprograms-limitations- Steps for creating fuzzy logic tool box -GUI editor- rule viewer -membership function editor rule editor.

CREATING A SIMPLE MODEL IN SIMULINK

You can use Simulink® to model a system and then simulate the dynamic behavior of that system. Simulink allows you to create block diagrams, where blocks you connect represent parts of a system, and signals represent input/output relationships between those blocks. The primary function of Simulink is to simulate behavior of system components over time. In its simplest form, this task involves keeping a clock, determining the order in which the blocks are to be simulated, and propagating the outputs, computed in the block diagram, to the next block. Consider a switch that turns on a heater. At each time step, Simulink must compute the output of the switch, propagate it to the heater, and then compute the heat output.

Often, the effect of a component's input on its output is not instantaneous. For example, turning on a heater does not result in an instant change in temperature. Rather, this action provides input to a differential equation, and the history of the temperature (a *state*) is also a factor. When the simulation of a block diagram requires solving a differential or difference equation, Simulink employs memory and numerical solvers to compute the state values for the time step.

Simulink handles data in three categories:

- Signals — Block inputs and outputs, computed during simulation
- States — Internal values, representing the dynamics of the block, computed during simulation
- Parameters — Values that affect the behavior of a block, controlled by the user

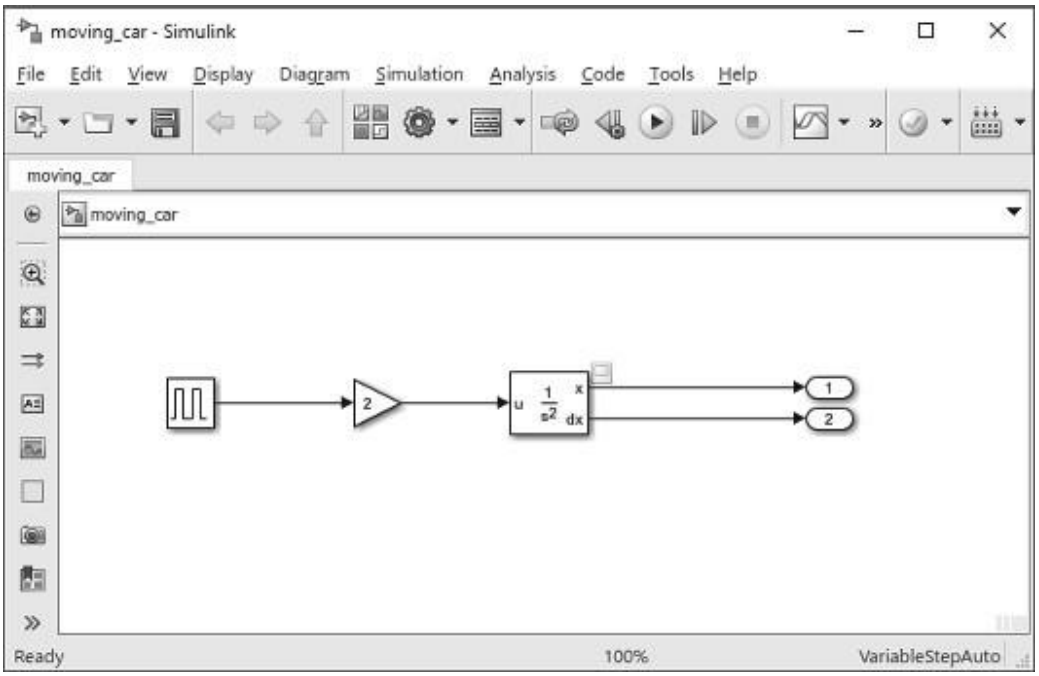
At each time step, Simulink computes new values for signals and states. By contrast, you specify parameters when you build the model and can occasionally change them while simulation is running.

Model Overview

The basic techniques you use to create a simple model in this tutorial are the same techniques that you use for more complex models. This example simulates simplified motion of a car, after a brief press of the accelerator pedal.

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

Block name	Block Purpose	Model Purpose
Pulse Generator	Generate an input signal for the model	Simulate the accelerator pedal
Gain	Multiply the input signal by a factor	Simulate how pressing the accelerator affects the car's acceleration
Integrator, Second-Order	Integrate input signal twice	Obtain position from acceleration
Outport	Designate a signal as an output from the model	Designate the position as an output from the model

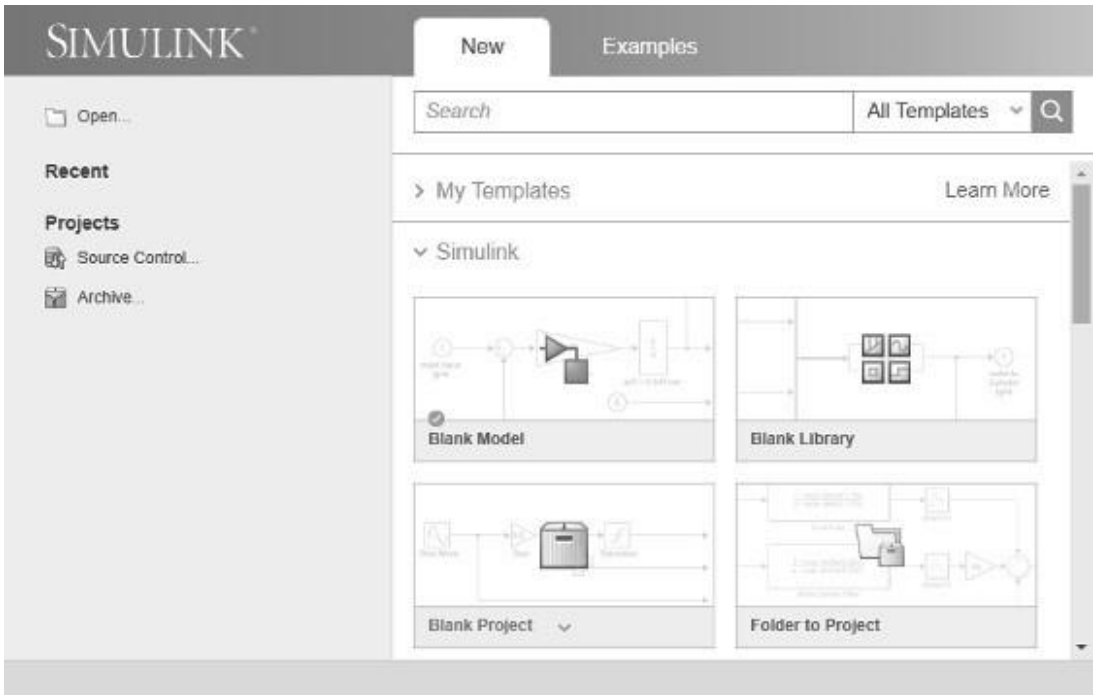


Simulating this model integrates a brief pulse twice to get a ramp and then displays the result in a Scope window. The input pulse represents a press of the accelerator pedal in a car, and the output ramp represents the increasing distance from the starting point.

Open New Model

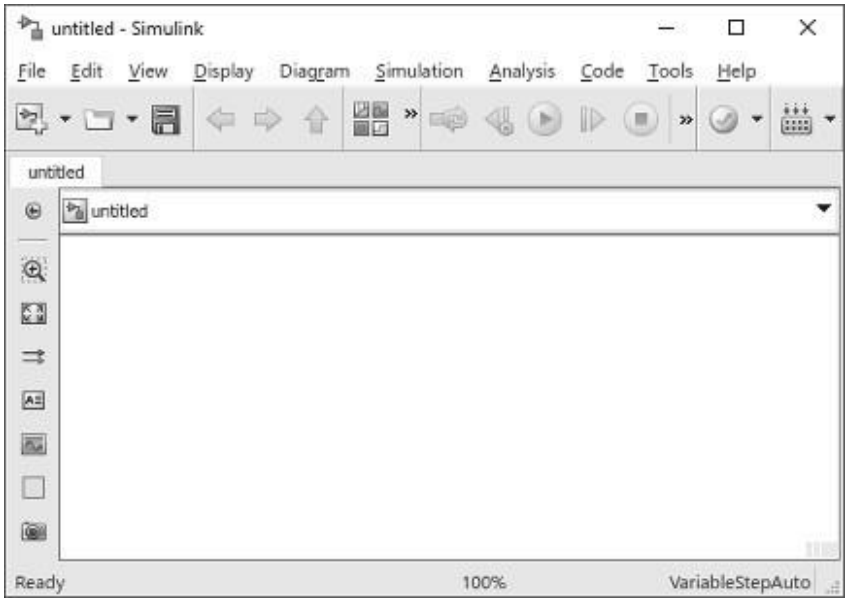
Use the Simulink Editor to build your models.

1. Start MATLAB®. From the MATLAB Toolstrip, click the **Simulink** button  .



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

   The Simulink Editor opens.



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

Open Simulink Library Browser

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

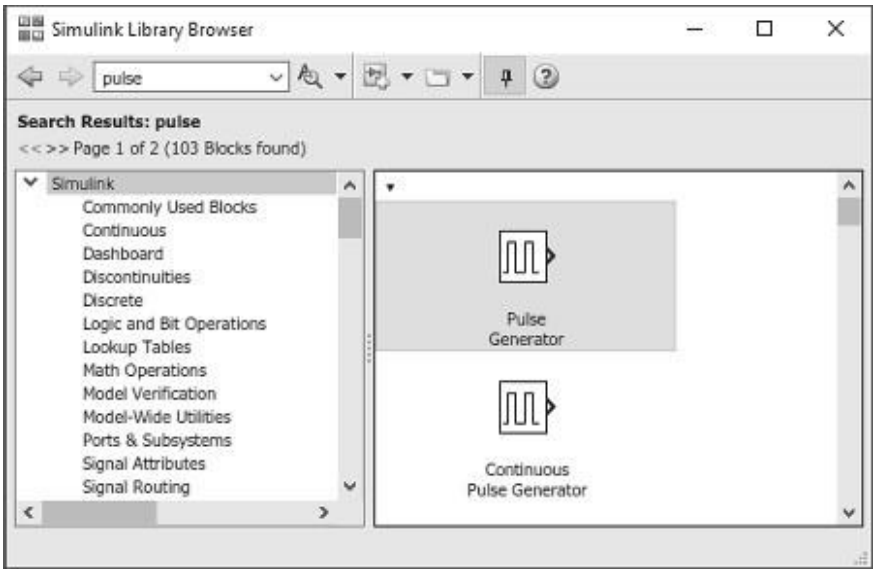
- Continuous — Building blocks for systems with continuous states
- Discrete — Building blocks for systems with discrete states
- Math Operations — Blocks that implement algebraic and logical equations
- Sinks — Blocks that store and show the signals that connect to them
- Sources — Blocks that generate the signal values that drive the model

1. From the Simulink Editor toolbar, click the **Library Browser** button  .

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

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

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



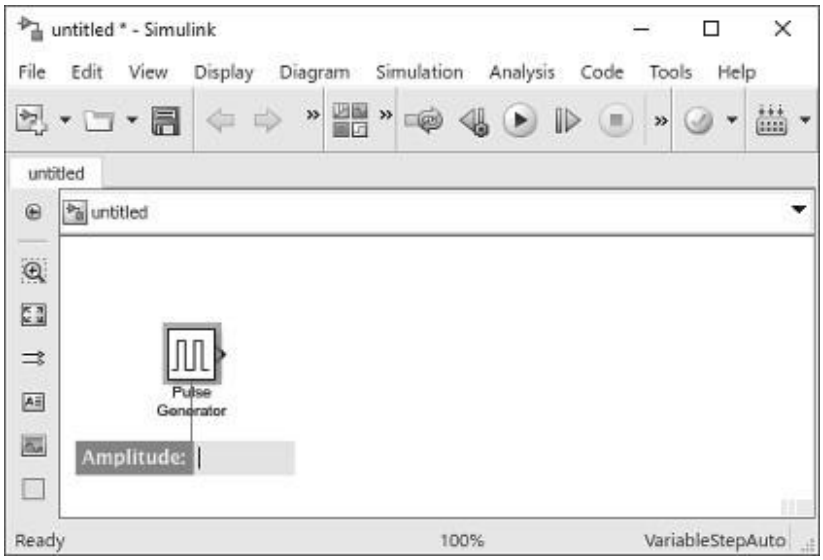
Get detailed information about a block. Right-click a block, and then select **Help for the Pulse Generator block**. The Help browser opens with the reference page for the block.

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

Add Blocks to a Model

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

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

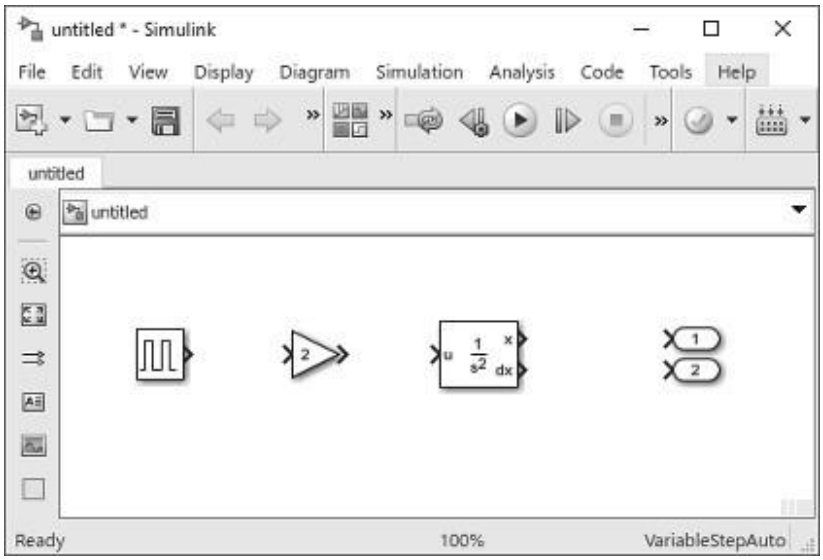


Parameter values are held throughout the simulation.

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

Block	Library
Gain	Simulink/Math Operations
Integrator, Second Order	Simulink/Continuous
Outport	Simulink/Sinks

3. Add a second Outport block right-clicking and dragging the existing one.
4. Your model should now have the blocks you need.
5. Arrange the blocks as follows by clicking and dragging each block. To resize a block, click and drag a corner.

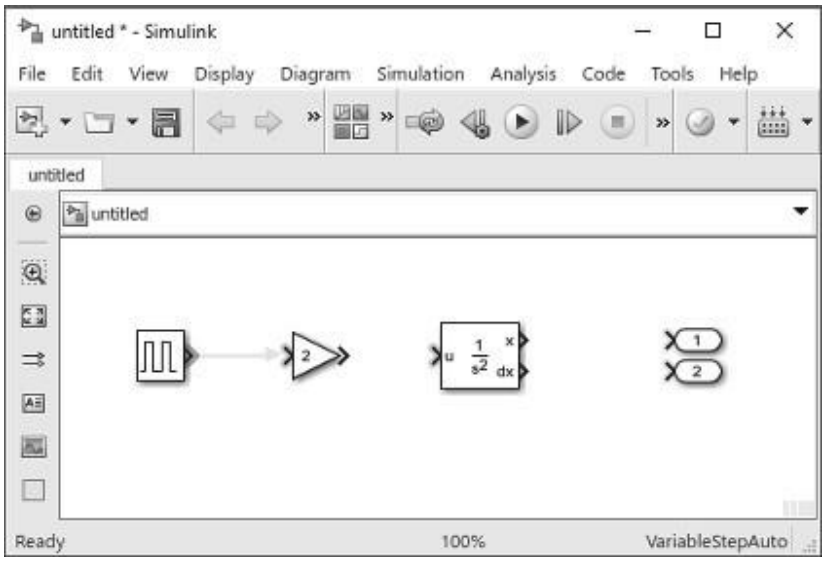


Connect Blocks

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

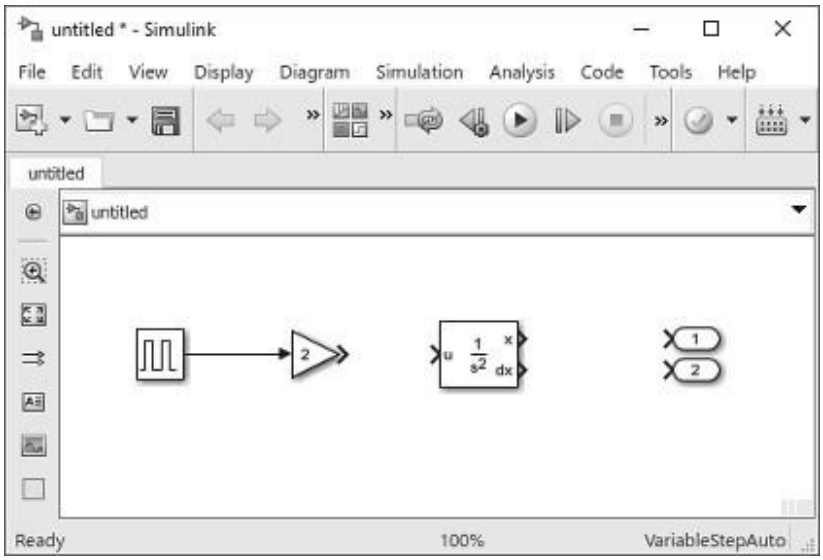
1. Click the output port on the right side of the Pulse Generator block.

   The output port, and all input ports suitable for a connection get highlighted.

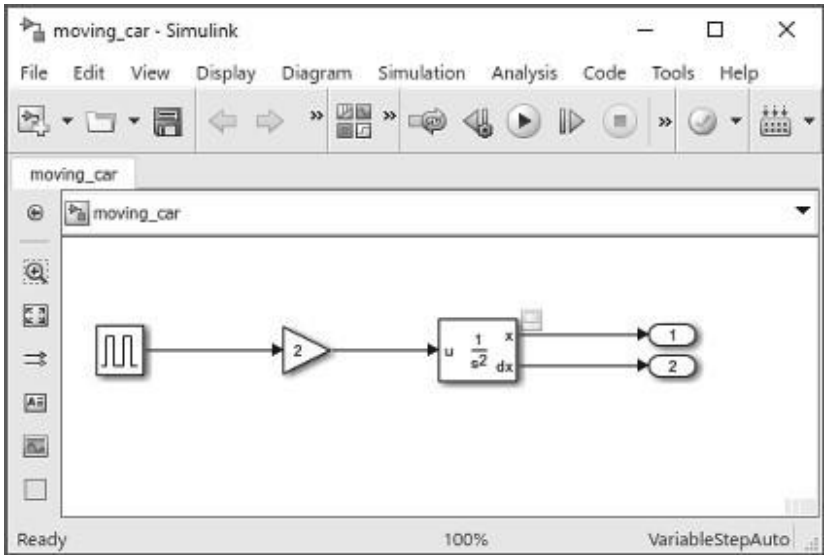


2. Click the input port of the Gain block.

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



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

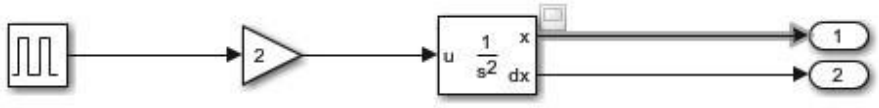


Your model is complete.

Add Signal Viewer

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

Access the context menu by right-clicking the signal. Select **Create & Connect Viewer > Simulink > Scope**. This creates a viewer icon on the signal, and opens a Viewer display.



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

Run Simulation

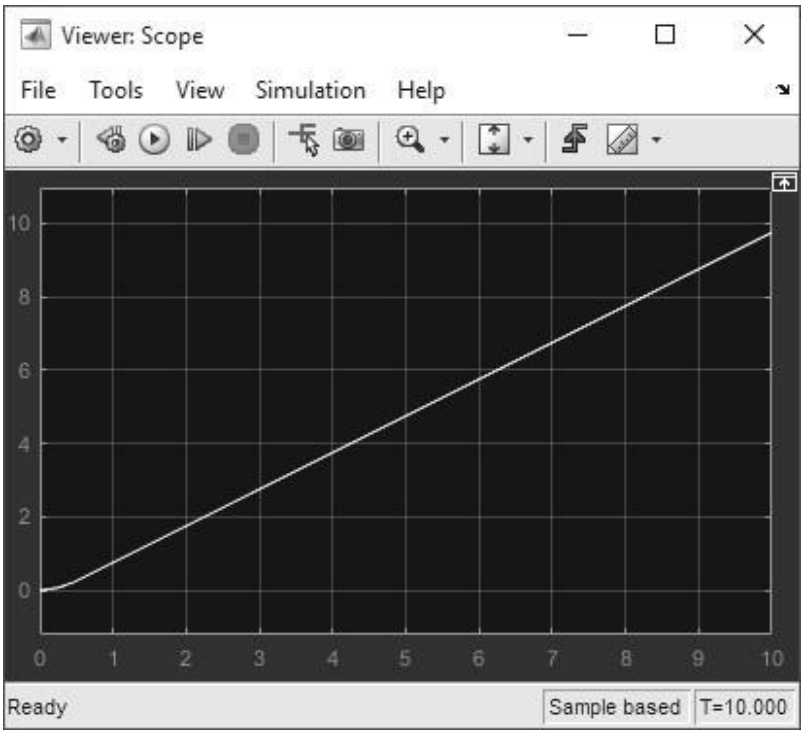
After you define the configuration parameters, you are ready to simulate your model.

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

The default stop time of 10.0 is appropriate for this model. This time value has no unit. Time unit in Simulink depends on how the equations are constructed. This example simulates the simplified motion of a car for 10 seconds.

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

The simulation runs and produces the output on the Viewer.



GENERATING AM IN SIMULINK

For generating AM we just have to implement the equation of AM in block level.

Blocks Required

Analyzing the equation we need,

1. Carrier Signal Source
2. Message Signal Source
3. Blocks for viewing the signals – Scope
4. Product Block
5. Summer Block
6. Constant Block

Carrier, Message, Constant blocks

- Simulink ➢ Sources ➢ Sine wave
- Simulink ➢ Sources ➢ Constant

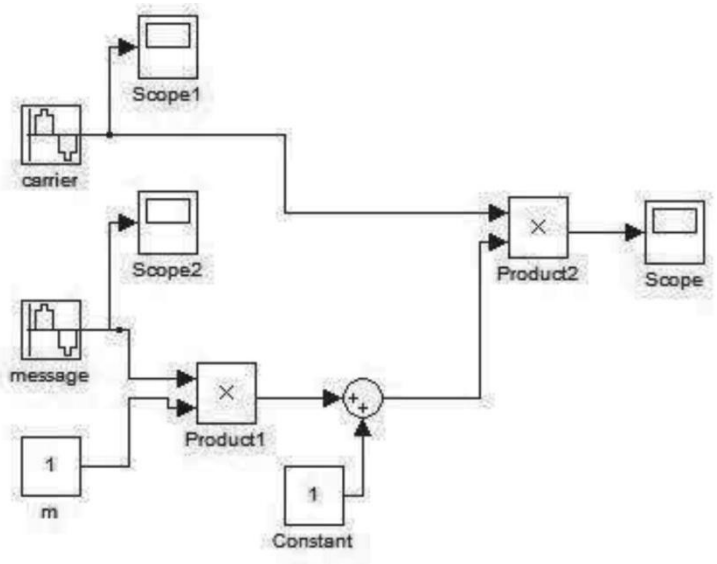
View Block

- Simulink ➢ Sink ➢ Scope

Product and Summer Block

- Simulink ➢ Math Operations ➢ Product
- Simulink ➢ Math Operations ➢ Summer

Block Diagram

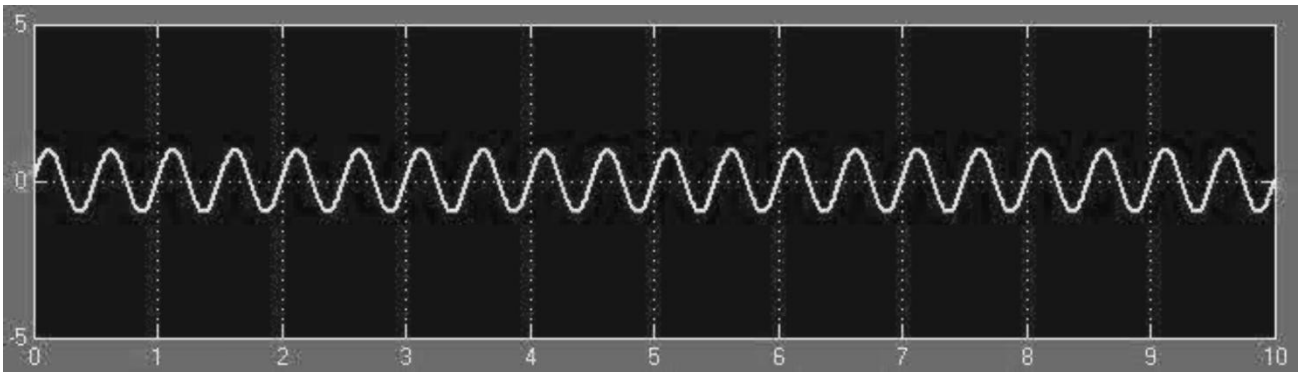


AM Generation using Simulink – Block Diagram

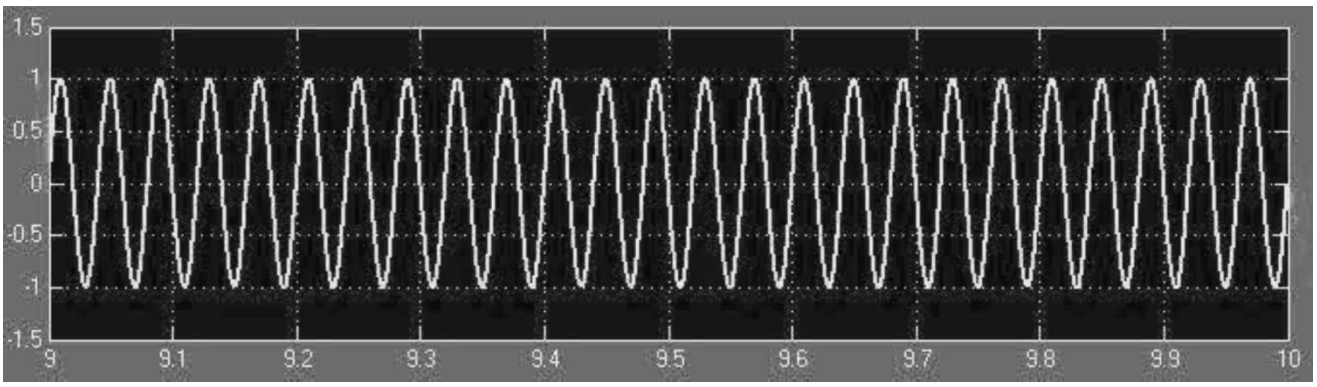
Block parameters can be changed by selecting the block and parameter that I used are given below..

- Carrier Signal frequency = 2*pi*25 and sampling time=1/5000
- Message Signal frequency = 2*pi and sampling time=1/5000
- Amplitudes of both signals are 1

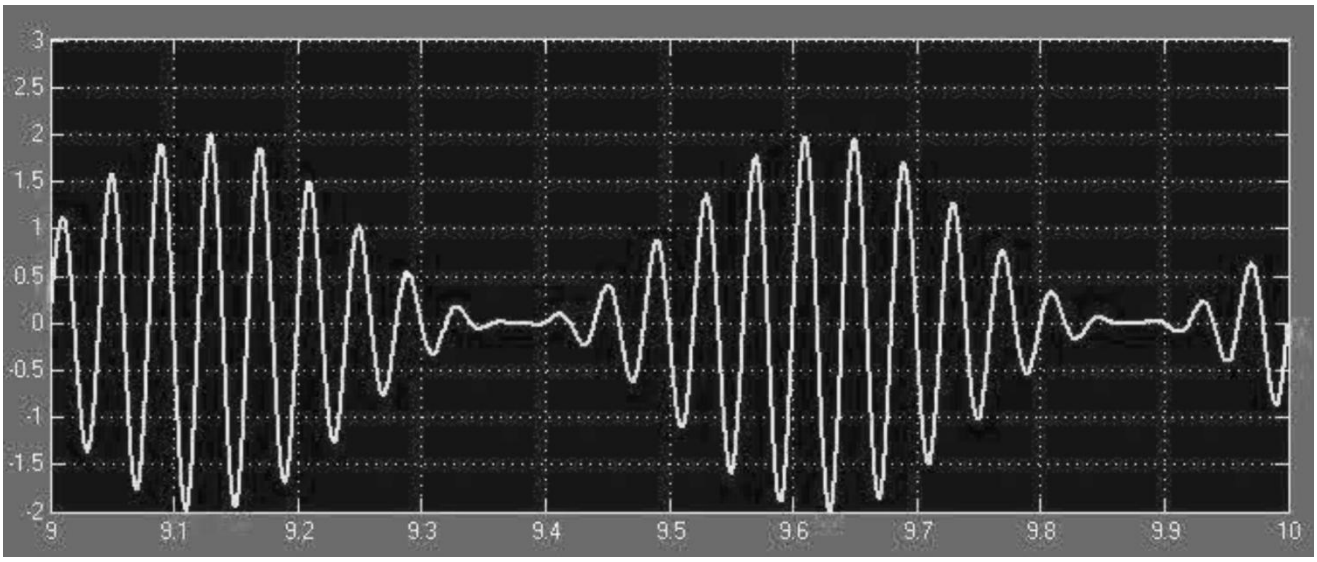
Output Waveforms



AM Generation using Simulink – Message Signal



AM Generation using Simulink – Carrier



AM Generation using Simulink – Modulated Signal

GENERATING SECOND ORDER SYSTEM RESPONSE IN SIMULINK

To obtain the step response of a 2nd order system for both open and closed loop.

Blocks Required

- Step input
- Transfer Function
- PID Controller
- Summer block
- Scope

Step input

- Simulink ≥ Sources > Step

View output

- Simulink ≥ Sink > Scope

Summer Block

- Simulink ≥ Math Operations > Summer

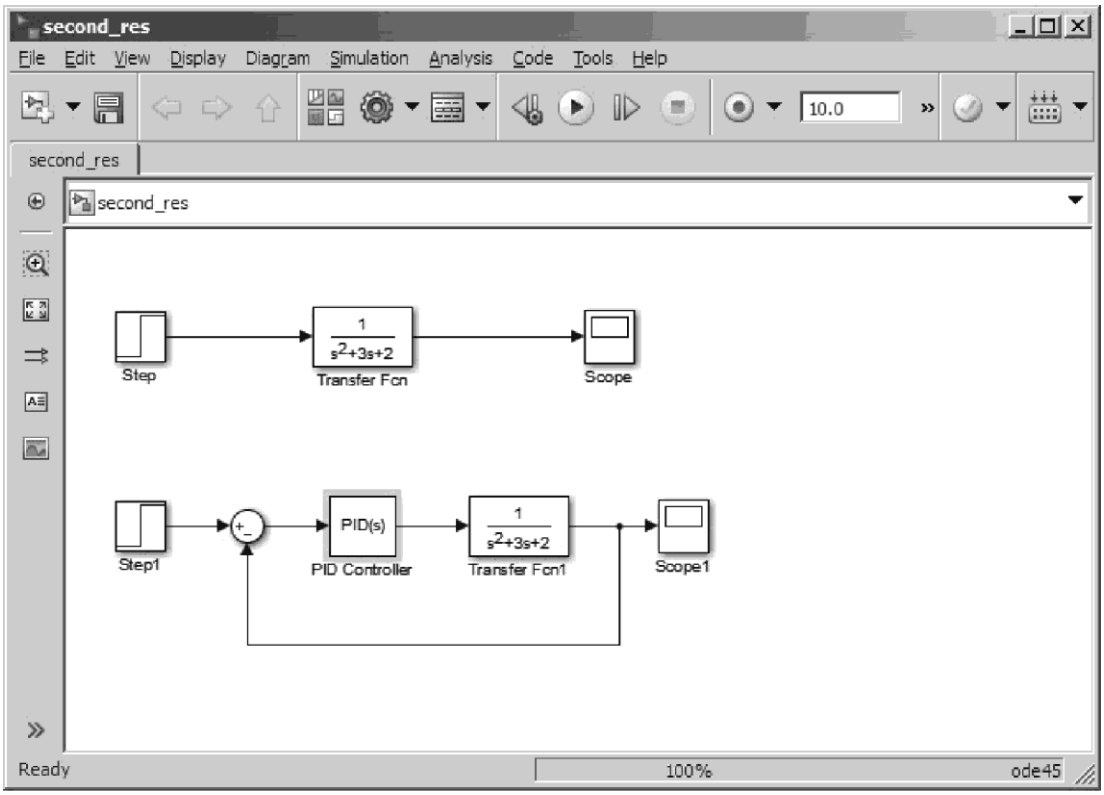
Transfer Function

- Simulink ≥ Continuous > Transfer function

PID Controller

- Simulink ≥ Continuous > PID controller

Block Diagram of Open Loop system and closed loop System



Transfer Fcn

The numerator .tx›efFicient can be a vactor or matrix digression.. The deromiriator-coefficient must be a vector. 'The output width equals fhe number ot rows in.tl\e numerator coefficient. "You ahould spec'ifi/ tke coefficients‹ in déscerid"ing..order of powers off.

Parameters

Nurñérator éoefF<le,nts:

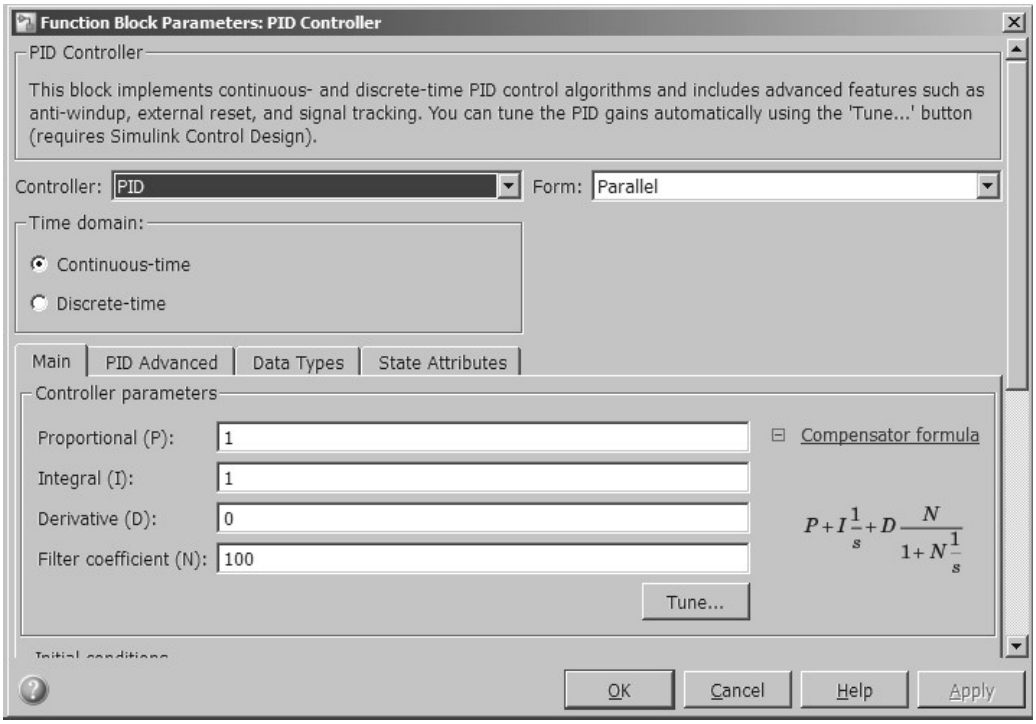
[132j

Absolute tolerance.

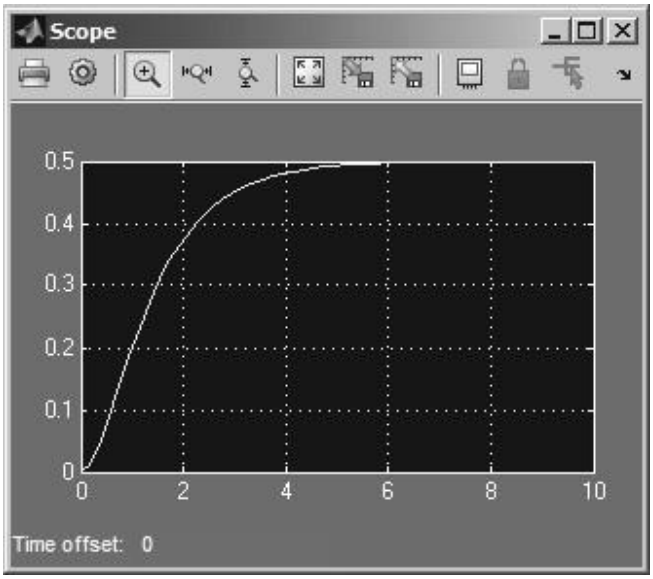
auto

'Staie i\lame:" {e.g., 'pbsiiiong

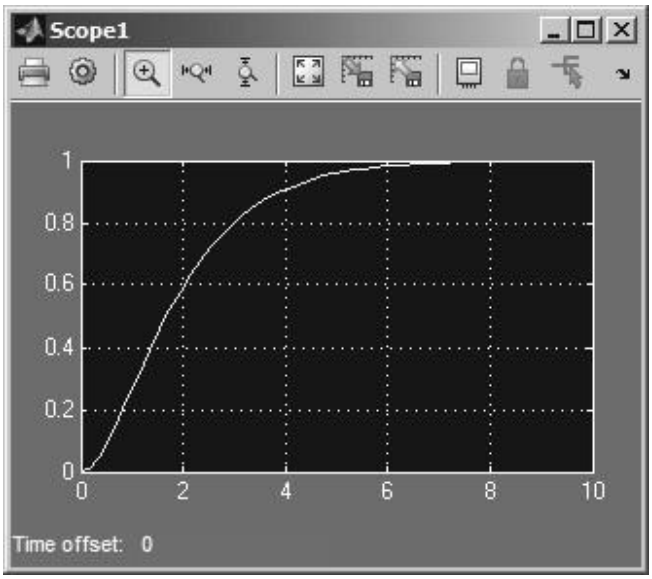
OK Eancef Help Apply



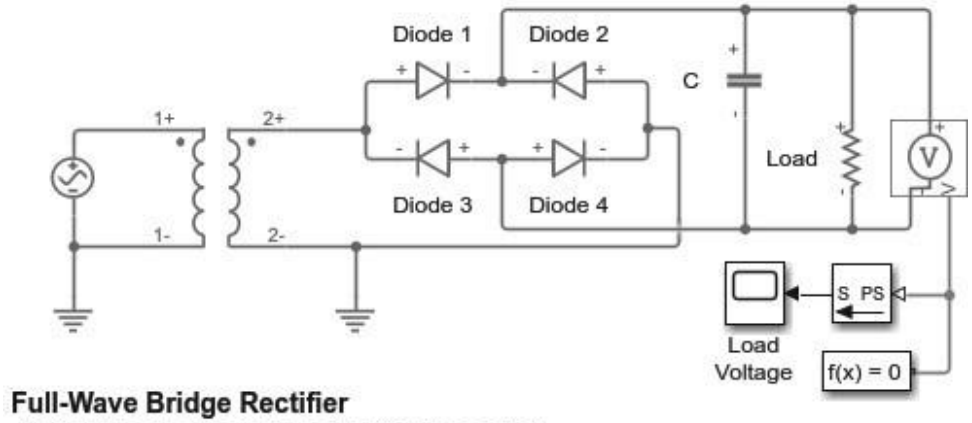
Open Loop Response



Closed Loop response



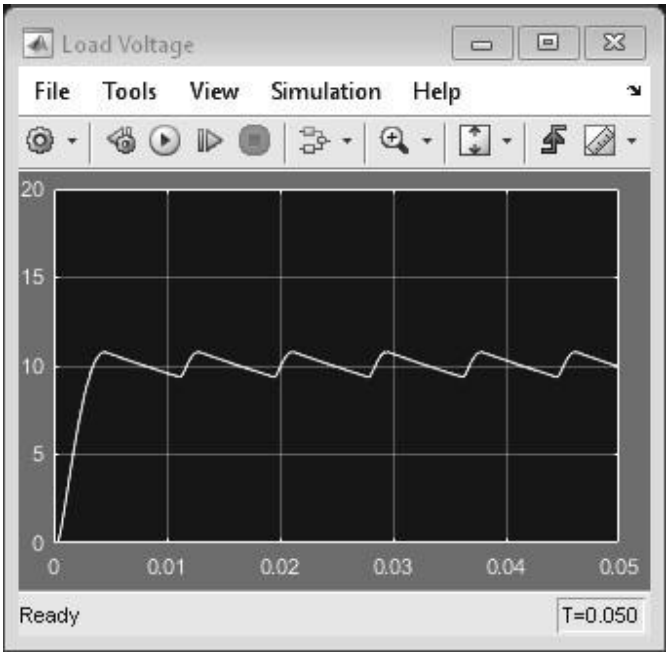
Full Wave Bridge Rectifier



Full-Wave Bridge Rectifier

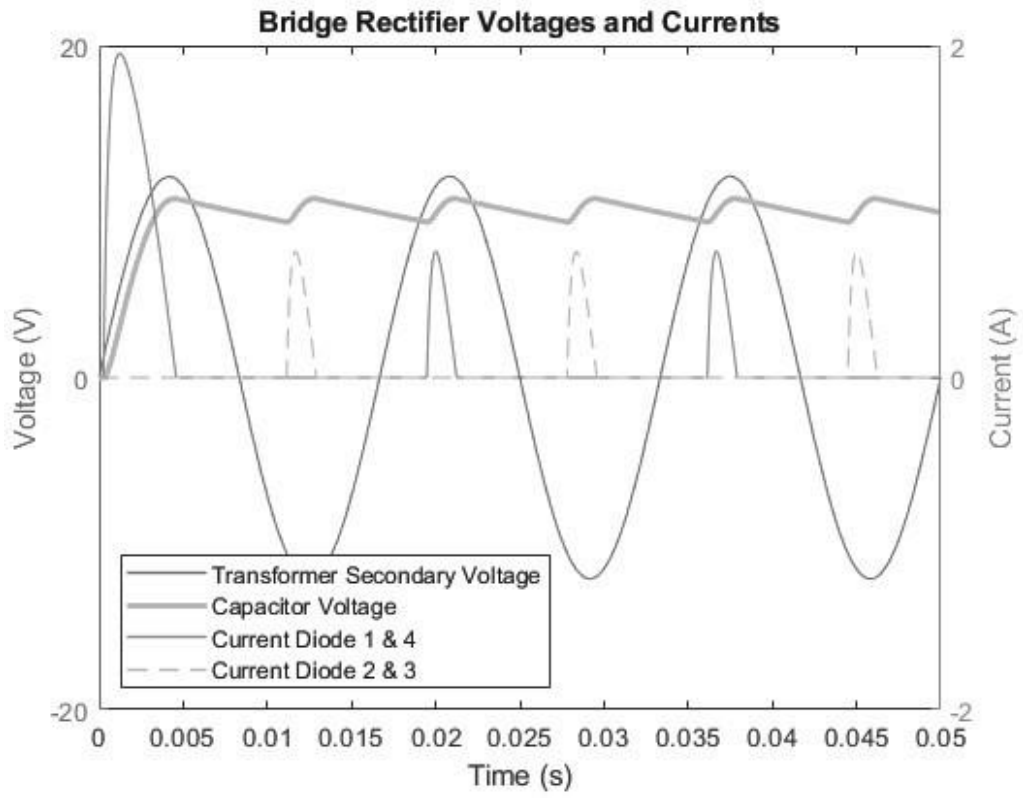
1. Plot voltages and currents of rectifier (see code)
2. Explore simulation results using sscexplore
3. Learn more about this example

Simulation Results from Scopes

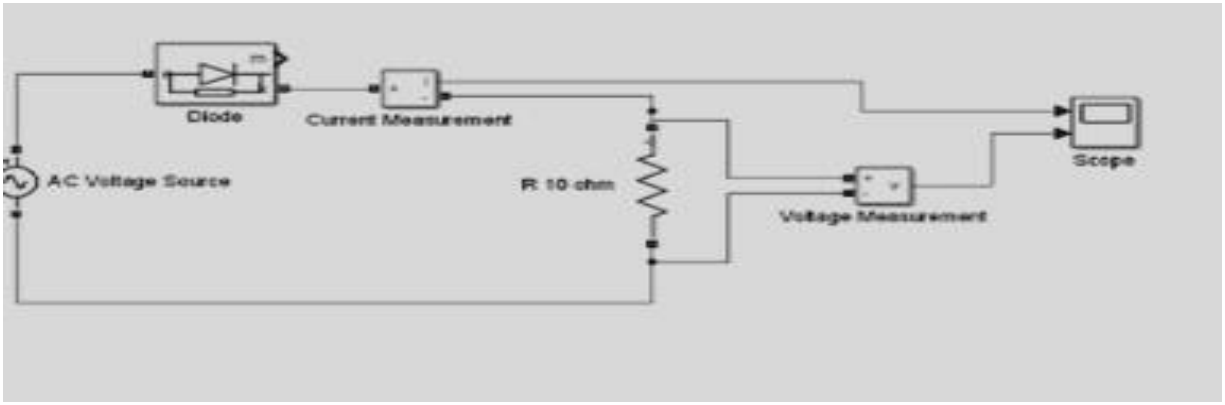


Simulation Results from Simscape Logging

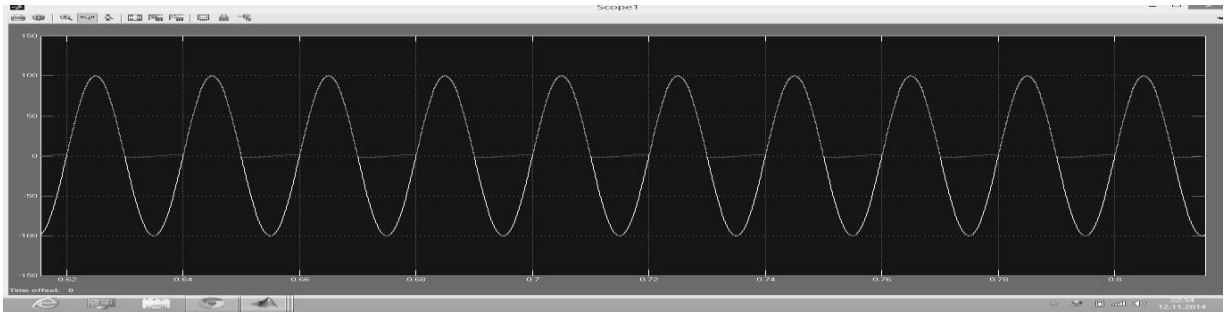
Plot "Bridge Rectifier Voltages and Currents" shows how AC voltage is converted to DC Voltage. The dark blue line is the AC voltage on the source side of the bridge. There are two paths for current flow through the diode bridge. The alternating peaks through diodes 1&4 and diodes 2&3 show that current flow reaching the capacitor is flowing in the same direction even though the polarity of the voltage is changing. The ripple in the load voltage corresponds to the charging and discharging of the capacitor.



In the same manner half wave rectifier can be designed using Simulink.



AC current source is used. The diode is connected in series to the Source. The Output is taken across the Load resister. Current and Voltage measurements can be seen through the Scope. The output contains only one half of the cycle of the input waveform. The below figure shows the output of a half wave rectifier.



Create a Subsystem

Subsystem Advantages

Subsystems allow you to create a hierarchical model comprising many layers. A subsystem is a set of blocks that you replace with a single Subsystem block. As your model increases in size and complexity, you can simplify it by grouping blocks into subsystems. Using subsystems:

- Establishes a hierarchical block diagram, where a Subsystem block is on one layer and the blocks that make up the subsystem are on another
- Keeps functionally related blocks together
- Helps reduce the number of blocks displayed in your model window

When you make a copy of a subsystem, that copy is independent of the source subsystem. To reuse the contents of a subsystem across a model or across models, use either model referencing or a library.

Ways to Create a Subsystem

You can create a subsystem using these approaches:

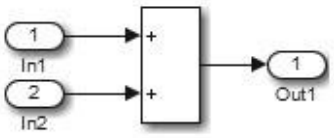
- Add a Subsystem block to your model, and then open the block and add blocks to the subsystem window. Create a Subsystem in a Subsystem Block.
- Select the blocks that you want in the subsystem, and from the right-click context menu, select **Create Subsystem from Selection**. Create a Subsystem from Selected Blocks.
- Copy a model to a subsystem. In the Simulink® Editor, copy and paste the model into a subsystem window, or use Simulink.BlockDiagram.copyContentsToSubsystem.
- Copy an existing Subsystem block to a model.
- Drag a box around the blocks you want in a subsystem, and select the type of subsystem you want from the context options. Create a Subsystem Using Context Options.

Create a Subsystem in a Subsystem Block

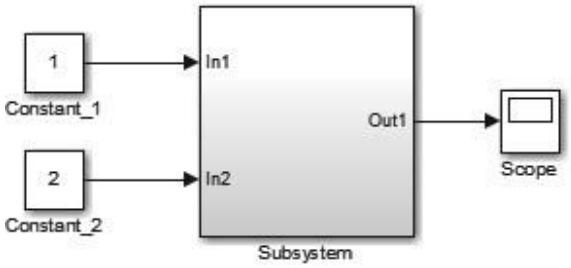
Add a Subsystem block to the model, and then add the blocks that make up the subsystem.

1. Copy the Subsystem block from the Ports & Subsystems library into your model.
2. Open the Subsystem block by double-clicking it.
3. In the empty subsystem window, create the subsystem contents. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output.

   For example, this subsystem includes a Sum block and Inport and Outport blocks to represent input to and output from the subsystem.



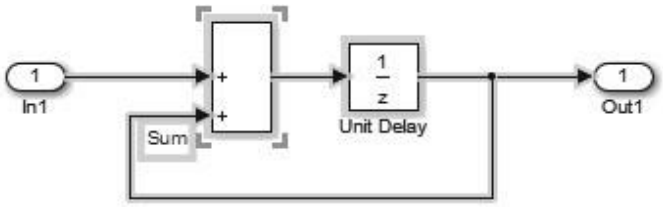
   When you close the subsystem window, the Subsystem block includes a port for each Inport and Outport block.



Create a Subsystem from Selected Blocks

1. Select the blocks that you want to include in a subsystem. To select multiple blocks in one area of the model, drag a bounding box that encloses the blocks and connecting lines that you want to include in the subsystem.

   The figure shows a model that represents a counter. The bounding box selects the Sum and Unit Delay blocks.

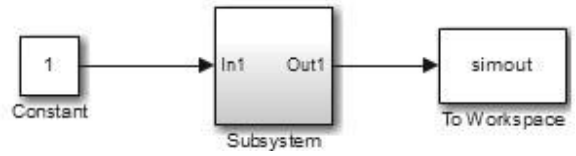


2. Select **Diagram** > **Subsystems & Model Reference** > **Create Subsystem from Selection**.

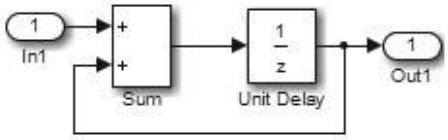
   A Subsystem block appears, which encloses the selected blocks.

Tip

Resize the Subsystem block so the port labels are readable.



To edit the subsystem contents, open the Subsystem block. For example:

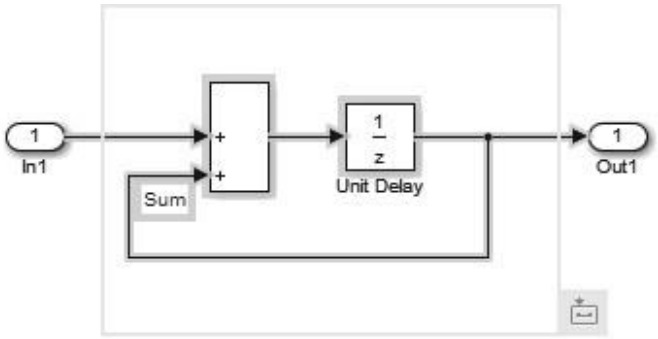


adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem.

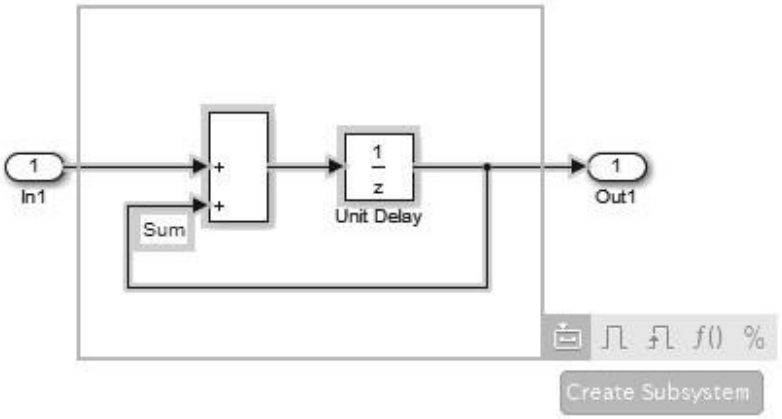
You can change the name of the Subsystem block and modify the block the way that you do with any other block (for example, you can mask the subsystem).

Create a Subsystem Using Context Options

1. Drag a box around the blocks you want in your subsystem.



2. View the subsystems you can create with these blocks by hovering over the first context option that appears.



3. Select the type of subsystem you want to create from these options.

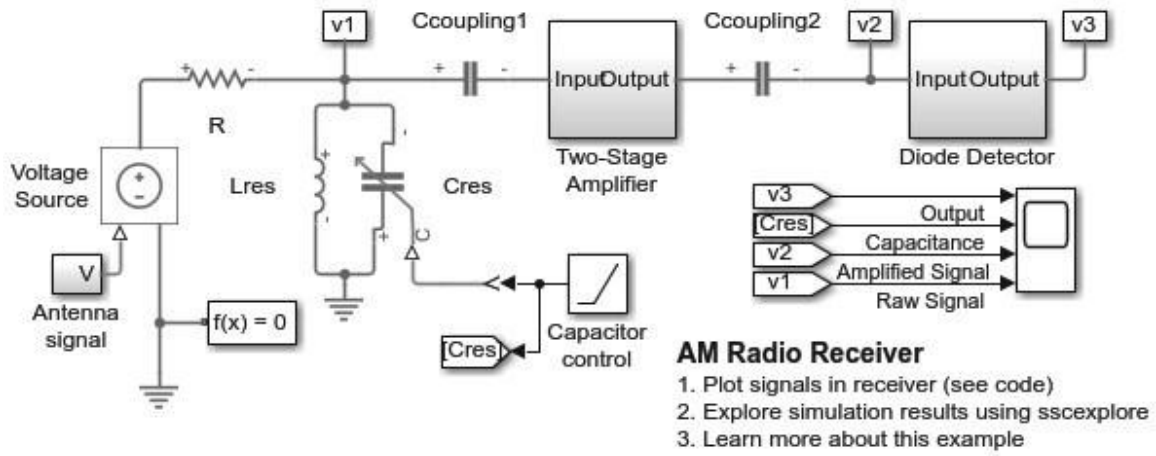
A Subsystem block appears, which encloses the selected blocks.

Few application programs in simulink

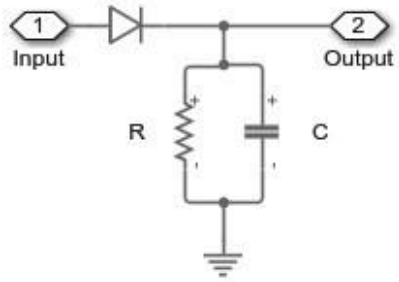
Example 1: AM Radio Receiver

This example shows a simplified AM radio receiver. A single tone signal at 2 kHz is transmitted with a carrier frequency of 600 kHz. The variable capacitor, Cres, in the resonant circuit is used in order to sweep through a certain frequency span. When the resonance passes through 600 kHz, the signal is picked up and amplified by a two-stage Class A RF power amplifier. The signal is finally extracted by a diode detector, where it would normally be passed on to an audio amplifier (not included here). The Scope displays the final output, the value of the resonant capacitance, and the received and amplified signals.

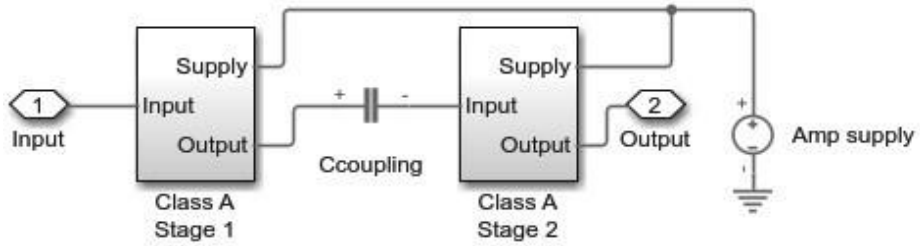
Model



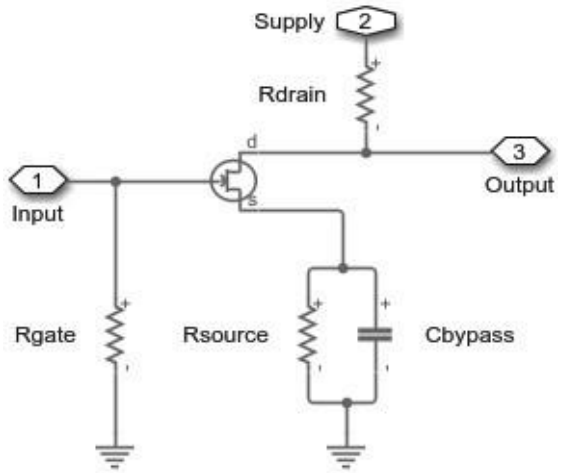
Diode Detector Subsystem



Two-Stage Amplifier Subsystem

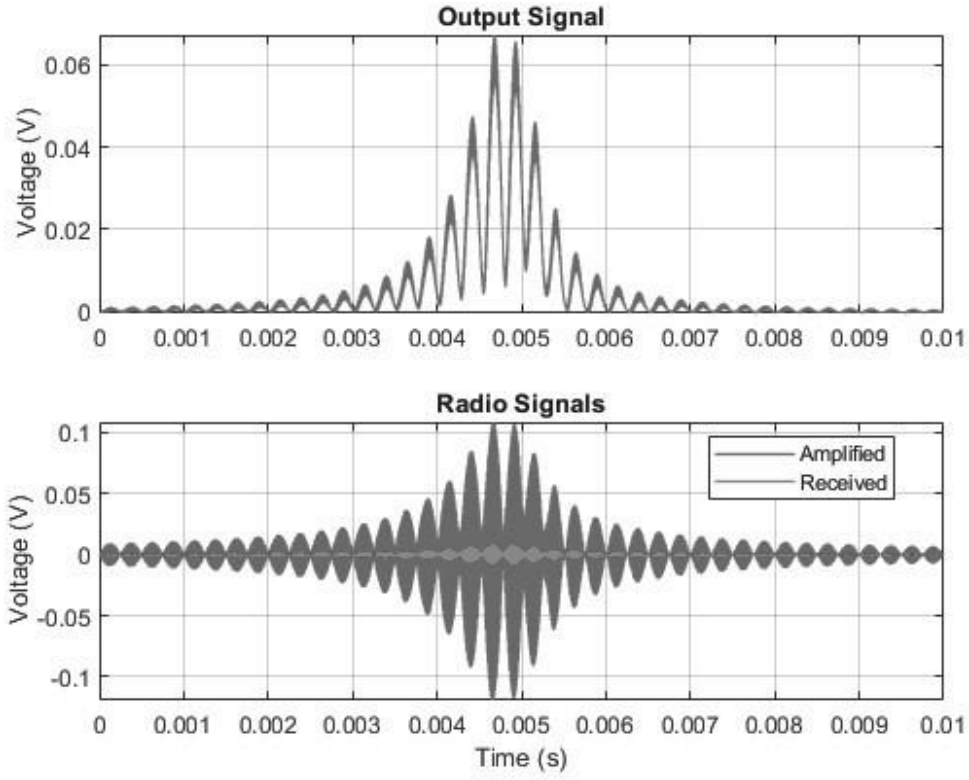


Class A Stage 1 Subsystem



Simulation Results from Simscape Logging

The plots below shows received, amplified, and output signals in the radio receiver. As the resonance in the resonant circuit passes through 600 kHz, the signal is picked up and amplified by a two-stage Class A RF power amplifier.



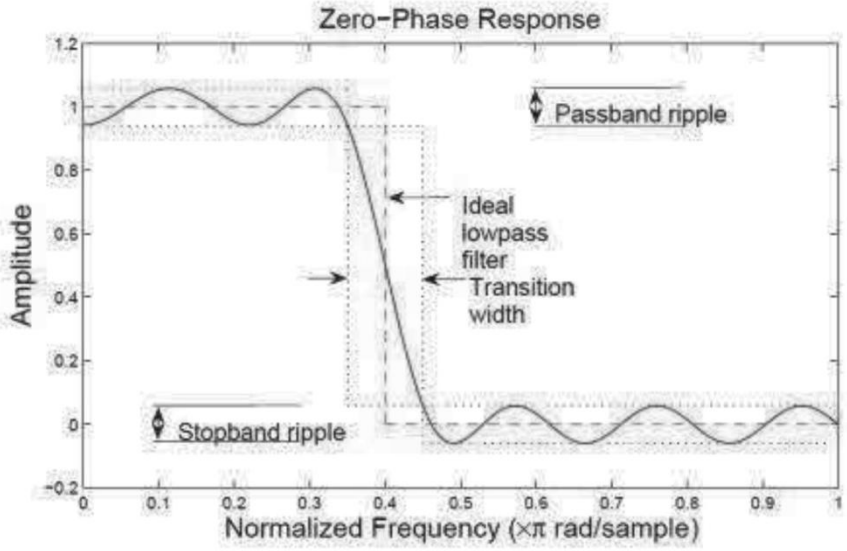
Example 2: FIR and IIR Filter Design and their Frequency Responses

This example shows how to design FIR and IIR filters based on frequency response specifications using the designfilt function in the Signal Processing Toolbox® product. The example concentrates on low pass filters but most of the results apply to other response types as well. And also focuses on the design of digital filters rather than on their applications..

Low pass Filter Specifications

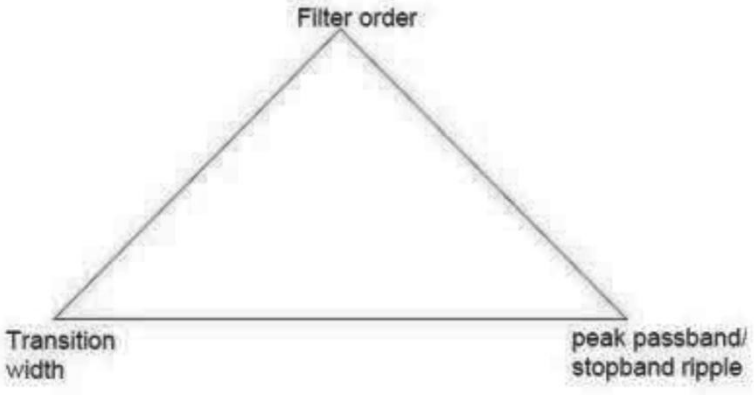
The ideal low pass filter is one that leaves unchanged all frequency components of a signal below a designated cutoff frequency ω_c, and rejects all components above ω_c. Because the impulse response required to implement the ideal low pass filter is infinitely long, it is impossible to design an ideal FIR low pass filter. Finite length approximations to the ideal impulse response lead to the presence of ripples in both the pass band ($\omega<\omega_c$) and the stop band ($\omega>\omega_c$) of the filter, as well as to a nonzero transition width between pass band and stop band.

Both the pass band/stop band ripples and the transition width are undesirable but unavoidable deviations from the response of an ideal low pass filter when approximated with a finite impulse response. These deviations are depicted in the following figure:



Practical FIR designs typically consist of filters that have a transition width and maximum pass band and stop band ripples that do not exceed allowable values. In addition to those design specifications, one must select the filter order, or, equivalently, the length of the truncated impulse response.

A useful metaphor for the design specifications in filter design is to think of each specification as one of the angles in the triangle shown in the figure below.



The triangle is used to understand the degrees of freedom available when choosing design specifications. Because the sum of the angles is fixed, one can at most select the values of two of the specifications. The third specification will be determined by the particular design algorithm. Moreover, as with the angles in a triangle, if we make one of the specifications larger/smaller, it will impact one or both of the other specifications.

FIR filters are very attractive because they are inherently stable and can be designed to have linear phase. Nonetheless, these filters can have long transient responses and might prove computationally expensive in certain applications.

Minimum-Order FIR Filter Design

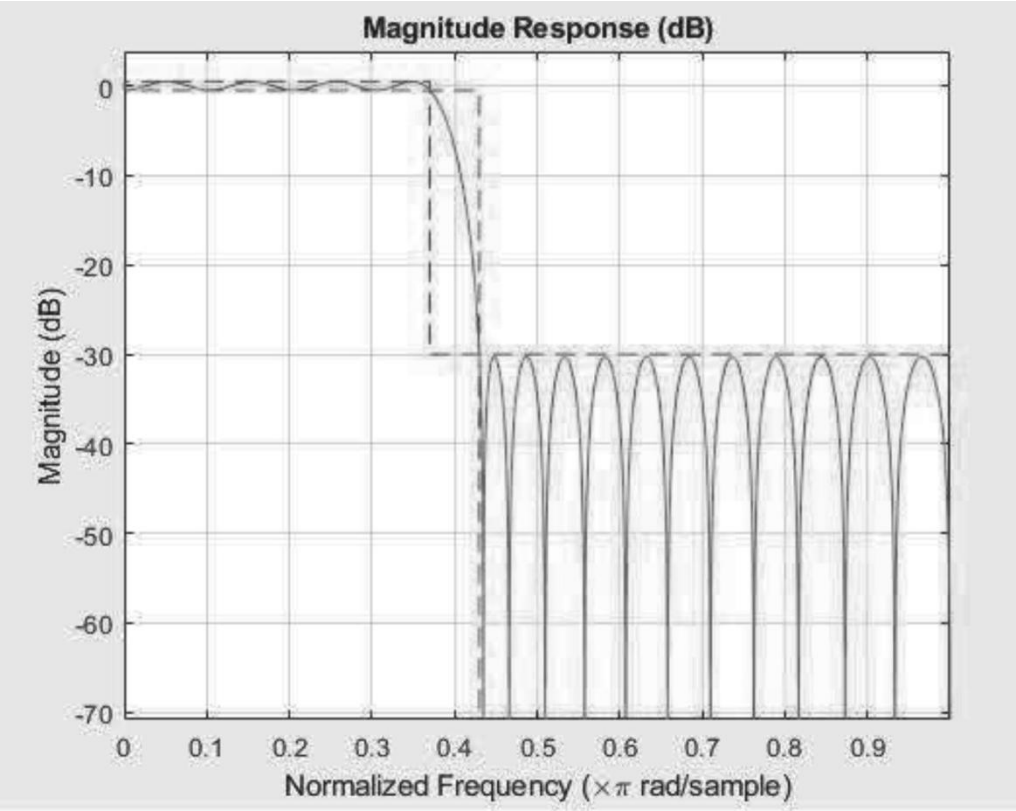
Minimum-order designs are obtained by specifying pass band and stop band frequencies as well as a pass band ripple and a stop band attenuation. The design algorithm then chooses the minimum filter length that complies with the specifications.

Design a minimum-order low pass FIR filter with a pass band frequency of 0.37*pi rad/sample, a stop band frequency of 0.43*pi rad/sample (hence the transition width equals 0.06*pi rad/sample), a pass band ripple of 1 dB and a stop band attenuation of 30 dB.

```
Fpass = 0.37;
Fstop = 0.43;
Ap = 1;
Ast = 30;


d = designfilt('lowpassfir','PassbandFrequency',Fpass,...
  'StopbandFrequency',Fstop,'PassbandRipple',Ap,'StopbandAttenuation',Ast);


hfvt = fvtool(d);
```



IIR Filter Design

One of the drawbacks of FIR filters is that they require a large filter order to meet some design specifications. If the ripples are kept constant, the filter order grows inversely proportional to the transition width. By using feedback, it is possible to meet a set of design specifications with a far smaller filter order. This is the idea behind IIR filter design. The term "infinite impulse response" (IIR) stems from the fact that, when an impulse is applied to the filter, the output never decays to zero.

IIR filters are useful when computational resources are at a premium. However, stable, causal IIR filters cannot have perfectly linear phase. Avoid IIR designs in cases where phase linearity is a requirement.

Another important reason for using IIR filters is their small group delay relative to FIR filters, which results in a shorter transient response.

Butterworth Filters

Butterworth filters are maximally flat IIR filters. The flatness in the pass band and stop band causes the transition band to be very wide. Large orders are required to obtain filters with narrow transition widths.

Design a minimum-order Butterworth filter with pass band frequency 100 Hz, stop band frequency 300 Hz, maximum pass band ripple 1 dB, and 60 dB stop band attenuation. The sample rate is 2 kHz.

```
Fp = 100;
```

FUZZY LOGIC TOOLBOX

GUI – EDITORS:

The following graphic user interface editors are available in the fuzzy logic tool box. Also the fuzzy logic tool box contains comprehensive tools to help us to automatic control, signal processing, system identification, pattern recognition, time series prediction, determining and functional application.

GUI – EDITORS are:

Fuzzy – basic FIS editor

MF edit – Membership function editor

Rule edit – Rule editor

Rule View – Rule Viewer & Fuzzy interface Algorithm

SUF view – Output surface viewer

Various membership function and command line FIS first are available in tool box.

Fuzzy (Basic FIS editor):

The FIS editor displays high level information about a fuzzy interface system at the top is a diagram of the system with each input to output Clearly tabulated by double clicking on the input or output boxes. We can bring up the member ship function. Double clicking on the fuzzy rule box of the diagram will bring up the rule editor.

MFEDIT (Membership function editor):

It is used to create, remove and modify the MFS. For a given fuzzy system, on the left side of the diagram is a variable pallete region that we use to select the current variable by clicking.

Rule Edit (Rule editor):

The surface displays the rules associate with a given fuzzy system. Rules can be edited and displayed in any one of the different modes.

Surf View:

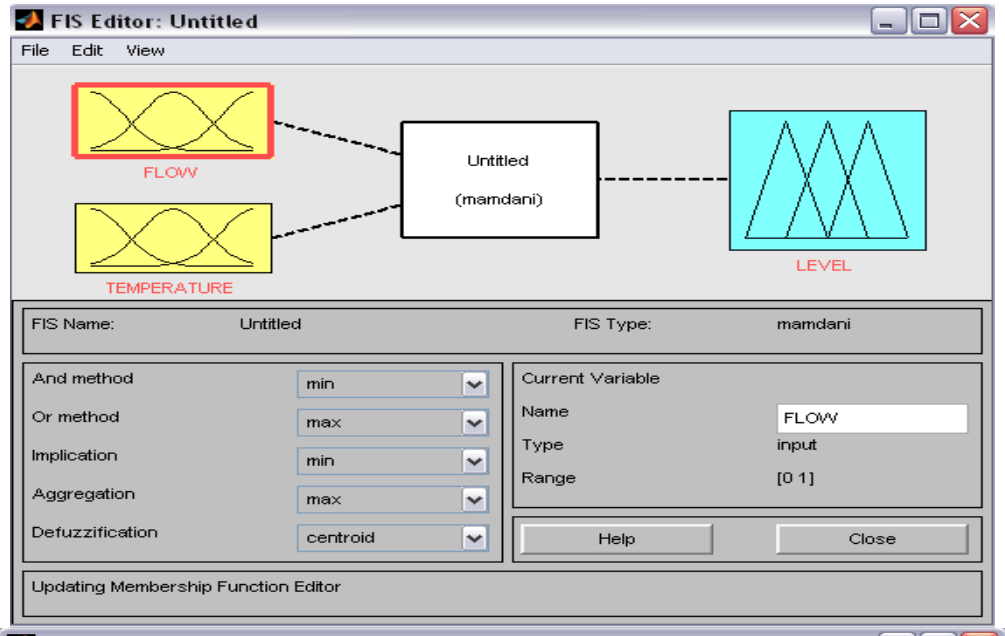
The surface viewer displays the entire output surface, output variables and input variables more than two input and output can be accommodated by using the "POP UP" menu.

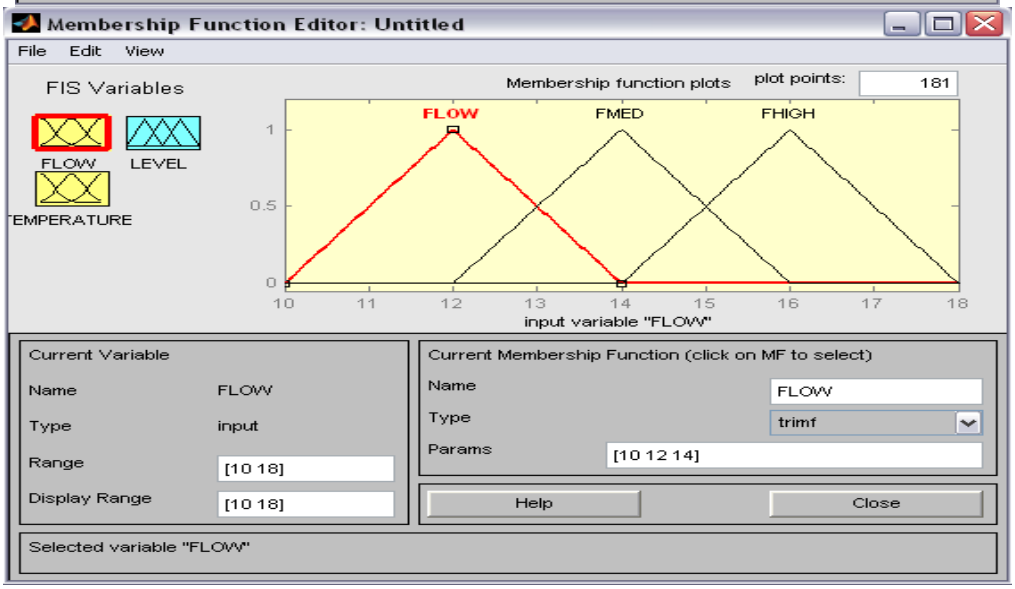
Problem:

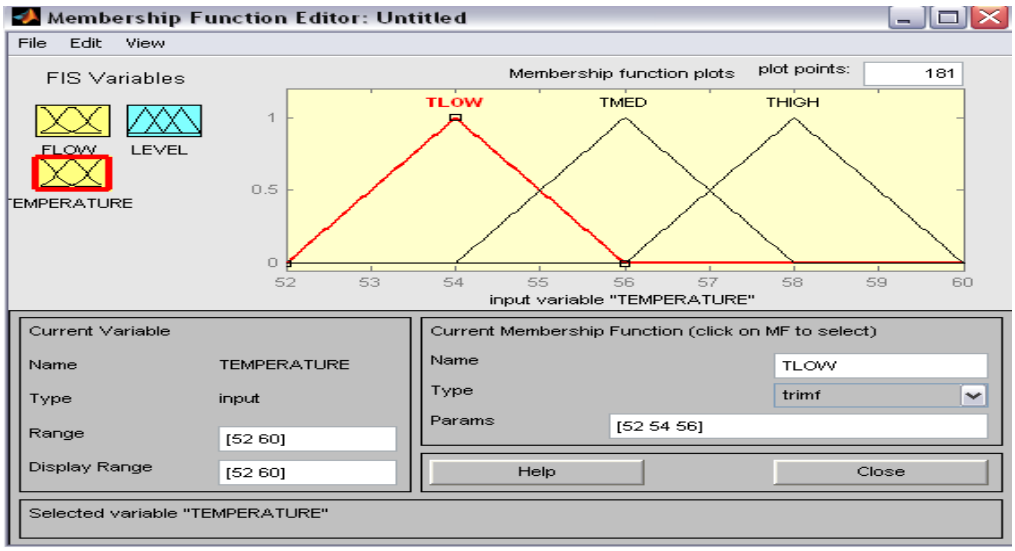
Design a fuzzy logic system to control the Level in a process line the Level to be maintained is between 14 to 18 m ,the inputs are the range of Flow is 10 to 18 l/s, the range of error is 52 to 60° C respectively.

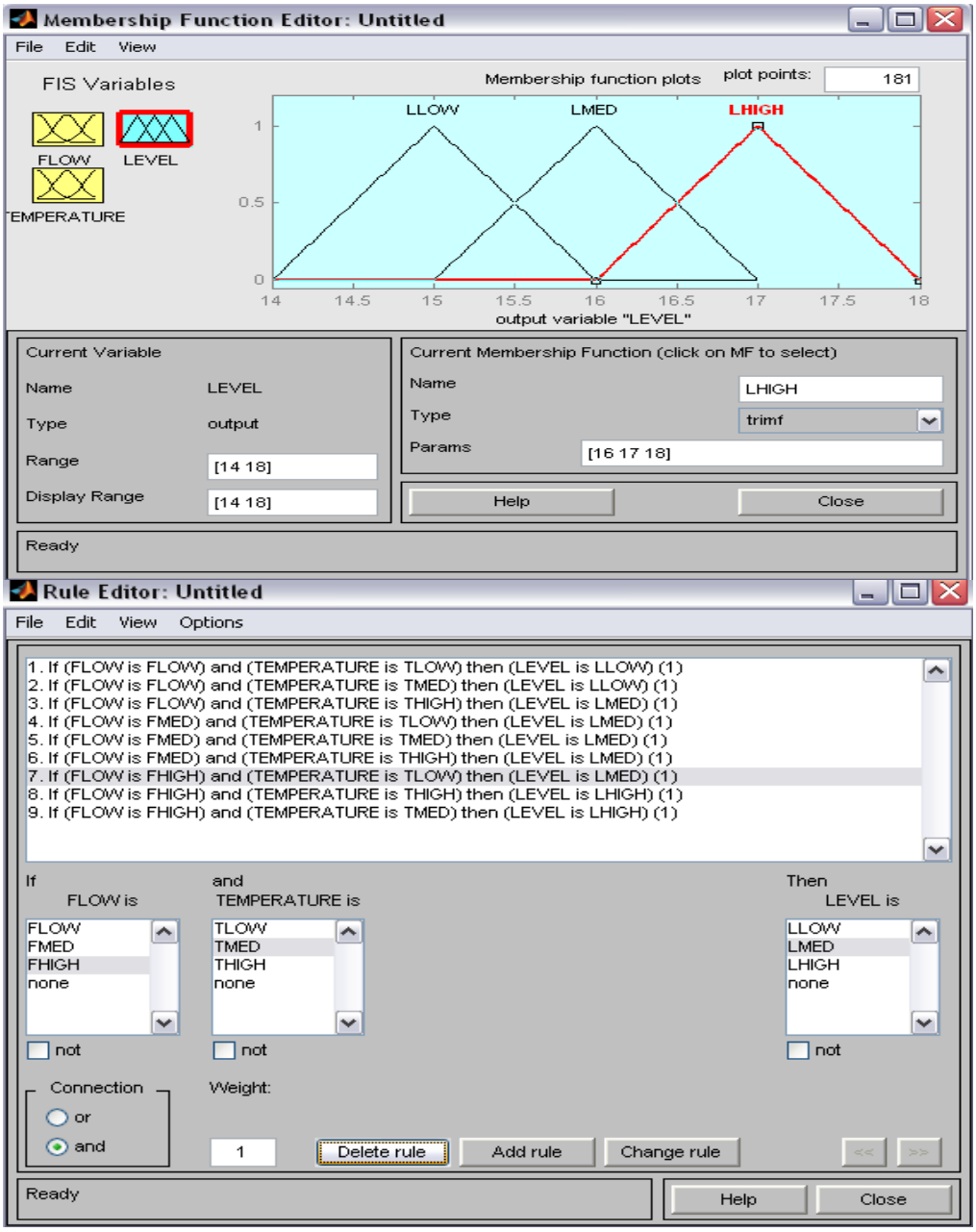
Procedure:

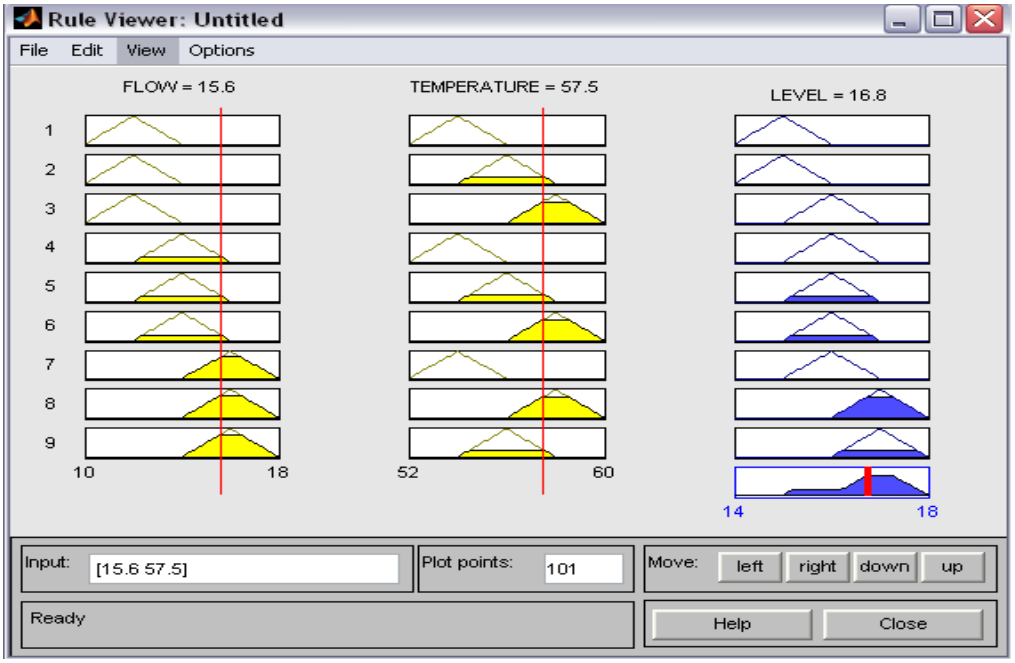
1. Open the Matlab Command window Enter FIS editor or fuzzy
2. In the FIS editor window , Edit-Add variable-input , to add the no. of inputs and outputs and enter the name.
3. Edit the membership function-range, type and its parameters. Create the rule editor –Add rule and view rules
4. Finally compare the results and write the inference.


FIS Editor: Untitled
File Edit View
FLOW
TEMPERATURE
Untitled
(mamdani)
LEVEL
FIS Name: Untitled
FIS Type: mamdani
And method min
Or method max
Implication min
Aggregation max
Defuzzification centroid
Current Variable
Name FLOW
Type input
Range [0 1]
Help
Close
Updating Membership Function Editor


Membership Function Editor: Untitled
File Edit View
FIS Variables
FLOW
LEVEL
TEMPERATURE
Membership function plots
plot points: 181
FLOW
FMED
FHIGH
input variable "FLOW"
Current Variable
Name FLOW
Type input
Range [10 18]
Display Range [10 18]
Current Membership Function (click on MF to select)
Name FLOW
Type trimf
Params [10 12 14]
Help
Close
Selected variable "FLOW"


Membership Function Editor: Untitled
File Edit View
FIS Variables
FLOW
LEVEL
TEMPERATURE
Membership function plots
plot points: 181
TLOW
TMED
THIGH
input variable "TEMPERATURE"
Current Variable
Name TEMPERATURE
Type input
Range [52 60]
Display Range [52 60]
Current Membership Function (click on MF to select)
Name TLOW
Type trimf
Params [52 54 56]
Help
Close
Selected variable "TEMPERATURE"


Membership Function Editor: Untitled
File Edit View
FIS Variables
FLOW
LEVEL
TEMPERATURE
Membership function plots
plot points:
181
LLOW
LMED
LHIGH
output variable "LEVEL"
Current Variable
Name
LEVEL
Type
output
Range
[14 18]
Display Range
[14 18]
Current Membership Function (click on MF to select)
Name
LHIGH
Type
trimf
Params
[16 17 18]
Help
Close
Ready
Rule Editor: Untitled
File Edit View Options
1. If (FLOW is FLOW) and (TEMPERATURE is TLOW) then (LEVEL is LLOW) (1)
2. If (FLOW is FLOW) and (TEMPERATURE is TMED) then (LEVEL is LLOW) (1)
3. If (FLOW is FLOW) and (TEMPERATURE is THIGH) then (LEVEL is LMED) (1)
4. If (FLOW is FMED) and (TEMPERATURE is TLOW) then (LEVEL is LMED) (1)
5. If (FLOW is FMED) and (TEMPERATURE is TMED) then (LEVEL is LMED) (1)
6. If (FLOW is FMED) and (TEMPERATURE is THIGH) then (LEVEL is LMED) (1)
7. If (FLOW is FHIGH) and (TEMPERATURE is TLOW) then (LEVEL is LMED) (1)
8. If (FLOW is FHIGH) and (TEMPERATURE is THIGH) then (LEVEL is LHIGH) (1)
9. If (FLOW is FHIGH) and (TEMPERATURE is TMED) then (LEVEL is LHIGH) (1)
If
FLOW is
FLOW
FMED
FHIGH
none
not
and
TEMPERATURE is
TLOW
TMED
THIGH
none
not
Then
LEVEL is
LLOW
LMED
LHIGH
none
not
Connection
or
and
Weight:
1
Delete rule
Add rule
Change rule
Ready
Help
Close


Rule Viewer: Untitled
File
Edit
View
Options
FLOW = 15.6
TEMPERATURE = 57.5
LEVEL = 16.8
1
2
3
4
5
6
7
8
9
10
18
52
60
14
18
Input:
[15.6 57.5]
Plot points:
101
Move:
left
right
down
up
Ready
Help
Close

REFERENCES

1. Stephen J Chapman, " Programming in MATLAB for Engineers", Brooks, 2002

2. Duane Hanselman ,Bruce LittleField, "Mastering MATLAB 7" , Pearson Education Inc, 2005

3. Stormy Attaway, "MATLAB a practical introduction to programming and problem solving", BH, 5th edt., 2001

4. Amos Gilat, "MATLAB an introduction with applications", Wiley, 2014

QUESTION BANK

PART A	CO
1. What is Simulink?	5
2. Mention the advantages of Simulink.	5
3. List the applications of Simulink.	5
4. How to create and run Simulink?	5
5. Name the command used for opening Simulink window in the command window	5
6. What is a subsystem?	5
7. Infer the significance of creating a subsystem.	6
8. Give the design procedure of any simple application in Simulink	6
9. Draw the Simulink model to get open loop gain of an OPAMP	6
10. List the standard test signals available in Simulink	6
PART B	
1. Create a Simulink application to generate AM Signal	5
2. How to create a subsystem? Explain with suitable example.	5
3. Create a Simulink application to generate PCM	5
4. Design a Fullwave and Halfwave rectifier using Simulink	6
5. Create a Simulink application to generate DPCM	6
6. Design a 2nd order system in Simulink and plot its open and close loop response.	5
7. Develop FIR filter and obtain the frequency response using MATLAB	5

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT – III – PROGRAMMABLE LOGIC CONTROLLERS– SEIA1503

Programmable logic controllers: Organization- Hardware details- I/O- Power supply- CPU Standards- Programming aspects-Ladder programming- Sequential function charts- Human-machine interface (HMI) - Case study on Lubrication System.

INTRODUCTION

PLC stands for Programmable Logic Controllers.

A PLC is basically a microprocessor based device that is used for controlling any machine (electrical, mechanical, and electronic). It is also used in assembly lines controlling in the industries. It is similar to a computer. It is typically based on RISC architecture. It is programmed in specific languages based on the real time purpose. It is connected to sensors, actuators, relays, contactors, etc. it is characterized by the number and type of I/O ports they provide and by their I/O scan rate

The National Electrical Manufacturers Association (NEMA) defines a PLC as a "digitally operating electronic apparatus which uses a programmable memory for the internal storage of instructions by implementing specific functions, such as logic, sequencing, timing, counting, and arithmetic to control through digital or analog I/O modules various types of machines or processes.

A PLC is able to receive (input) and transmit (output) various types of electrical and electronic signals and can control and monitor practically any kind of mechanical and/or electrical system. Therefore, it has enormous flexibility in interfacing with computers, machines, and many other peripheral systems or devices.

Control is the process in a system in which one or several input variables influence other variables.

Need for PLC

Hardwired panels were very time consuming to time, debug and change

The following requirements for computer controllers to replace hard wired panels:

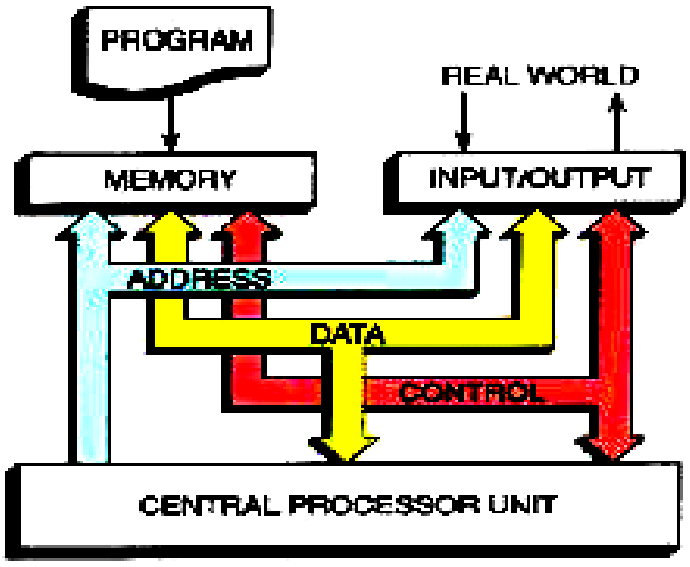
- Solid state not mechanical
- Easy to modify input and output devices
- Easily programmed and maintained by plant electricians
- Be able to function in an industrial environmentComparison

Comparison

Hard wired control systems	PLC systems
The functions are determined by physical wiring	The functions are determined by a program stored in the memory
Changing the function means changing the wiring	The control functions can be changed by simply changing the program
Can be contact making type (relays, contactors)or electronic type (logic circuits)	Consists of a control device, to which all the sensors and actuators are connected

PLC architecture

The basic architecture of the PLC can be described as below:



The basic components of a PLC are:

1. A Central Processing Unit (CPU)
2. Input module
3. Output module
4. Programming device
5. Power supply

Central Processing Unit:

It is the heart of the PLC system. The CPU is a microprocessor based control system that replaces central relays, counters, timers and sequencers. One bit processors are adequate for dealing with logic operations. The operations of a CPU are as follows:

- The CPU accepts data from various sensing devices, executes the user program from memory and sends appropriate output commands to control devices.
- A DC power source is required to produce a low -level voltage used by processor and I/O modules. This power supply can be housed in the CPU or may be a separately mounted unit, depending on the PLC system manufacturer.

Input and Output Modules:

Inputs:

Inputs are basically the physical entities that control the on and off of the machine. These devices are controlled by the either the machine operator manually or automatically sent after starting the machine. Some off the devices can be listed down as:

- Push buttons
- Limiter switches
- Pressure switches
- Flow switches
- Proxy sensor
- Emergency switch

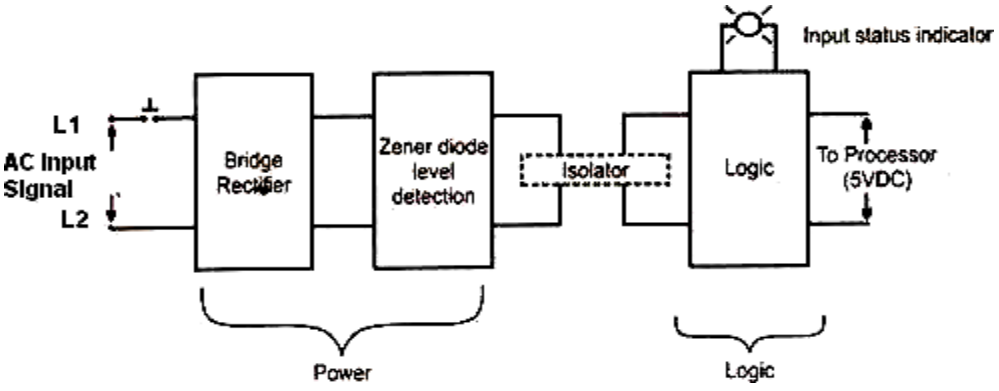
Outputs:

Outputs are the devices which receive the signals given by the PLC and perform the actions accordingly. The output devices can be listed down as:

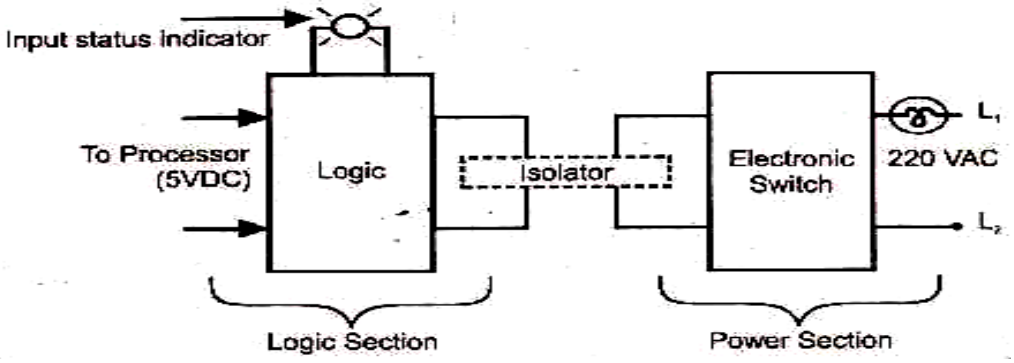
- Relays
- Contactors
- Solenoid valves
- Lamp indicators. Input-output module:

The I/O modules connect the input devices with the controller. They convert the electrical signals used in the devices into electronic signals that can be used by the control system and translate the real world values to IO table values.

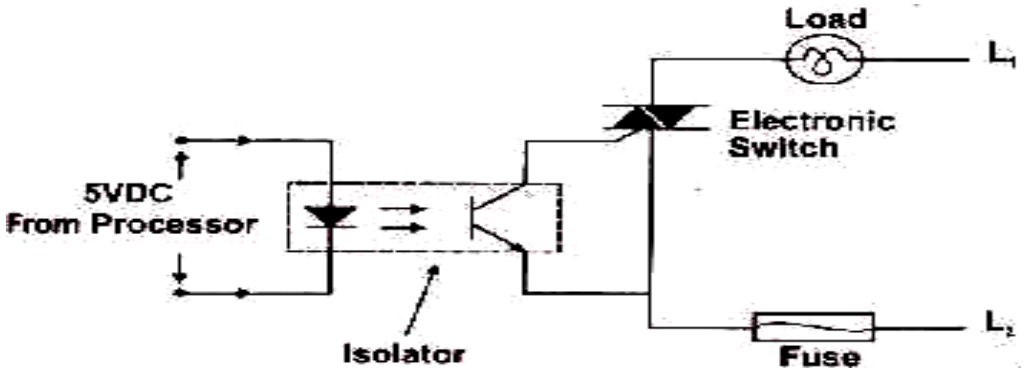
Block diagram of input module:



Block diagram of output module:



Circuit diagram of isolator circuit:



The number of I/O devices used within a control system is called its "point count".

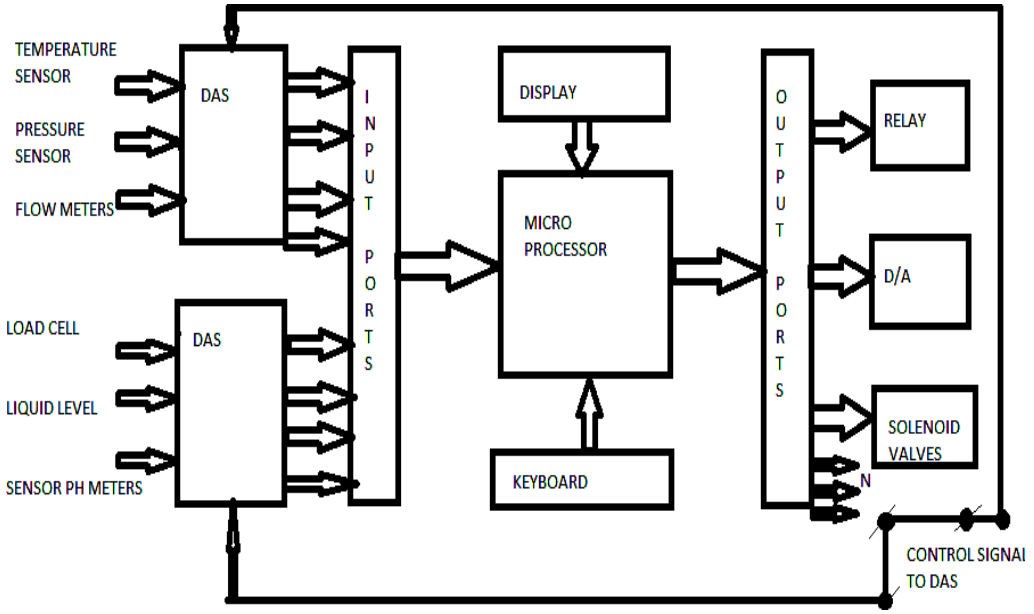
Programming device (keyboard and monitor):

Keyboard and monitor is used for programming the PLC. The data is entered in the processor with the help of the keyboard in the form of ladder diagram. This ladder diagram can be seen on the monitor screen. The programmer can communicate with the processor with the help of the programming devices. The programming unit communicates with the processor of the PLC via a serial or parallel data communication link.

PLC power supply:

The PLC is power by the AC mains supply. However some of the PLC components utilize power of about only 5V DC. PLC power supply converts the AC power supply into DC and supports these components.

The overall block diagram of the PLC:



The input signals are given to the **Data Acquisition System (DAS)** and then further sent to the input module. Further the signals are sent to the processor. The processor is connected with the programming devices. The signal from the processor is then sent to the output module and then from the output module further to the output devices.

PLC operating sequence

- Self- test: testing of its own hardware and software for faults.
- Input scan: if there are no problems, PLC will copy all the inputs and copy their values into memory.
- Logic solve/scan: using inputs, the ladder logic program is solved once and outputs are updated.
- Output scan: while solving logic, the output values are updated only in memory when ladder scan is done, the outputs will be updated using temporary values in memory.

Number systems and codes

The integral operation of programmable controllers is largely based on numbers. Numbers emulated by electronic circuitry are used to encode and store information that in turn se allow s these devices to process instructions and data required to perform each and every operation.

PLC also relies on number systems to perform even the most basic operations and store various information. The number systems commonly used in PLC are given below:

2	Binary	[0,1]
8	Octal	[0,1,2,3,4,5,6,7]
10	Decimal	[0,1,2,3,4,5,6,7,8,9]
16	Hexadecimal	[0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F]

In PLC, decimal number system is usually used for timer and counter presets, math operations and general numerical quantities.

Binary number system, bytes and words are the common units for storing digital information in memory. They are used to represent letters, numbers, punctuation marks, machine instructions and virtually any information type.

In PLC in many cases, base 8 is used for addressing memory and input/output terminal locations. Using octal for numbering is convenient in these cases since memory is comprised of groupings of 8 binary digits (bytes) and I /O modules are normally in one or more groups of eight points per module.

Hex numbers easily express coded digital data which is otherwise expressed in binary. Such codes are typically used when devices communicate. While the binary data may be easily interpreted by the receiving device, hex coded characters are really used for convenience of human operators.

Automation:

Automation is basically the delegation of human control functions to technical equipment aimed towards achieving higher productivity, superior quality of end product, efficient usage of energy and raw materials, improved safety in working conditions etc.

Advantages of PLC in Automation:

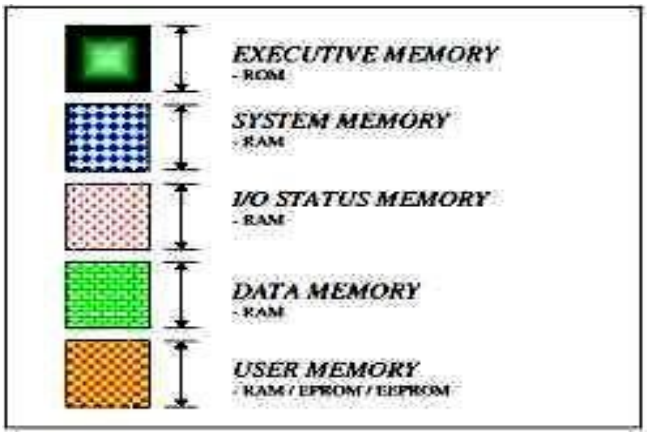
- To reduce human efforts
- To get maximum efficiency from machine and control them with human logic
- To reduce complex circuitry of entire system
- To eliminate the high costs associated with inflexible, relay controlled systems.

Areas of application of PLC in automation:

- Manufacturing/machining
- Food/beverage
- Textile industry
- Travel industry
- Aerospace
- Printing industry

Memory System:

All PLCs contain both RAM and ROM in varying amounts depending upon the design of the PLC. The use of PLC's memory is determined again by the design of the unit. However, all PLC memories can be subdivided into at least five major areas. A typical memory utilization map for a PLC is depicted in the following figure.



a. Executive Memory:

The operating system or executive memory for the PLC is always in ROM since, once programmed and developed by the manufacturer, it rarely needs changing. It is the one that actually does the scanning in a PLC. The operating system is a special machine language program that runs the PLC. It instructs the microprocessor to read each user instruction, helps the microprocessor to interpret user programmed symbols and instructions , keeps the track of all the I/O status, and is responsible for maintaining / monitoring the current status of the health of the system and all its components.

b. System memory

In order for the operating system to function, a section of the memory is allotted for system administration. As the executive program performs its duties, it often requires a place to store intermediate results and information. A section of RAM is installed for this purpose. Normally this area is allotted for use of the operating system only and is not available to the user for programming. It might be thought of as a scratch pad for the operating system to doodle on as necessary. Some PLCs use this area for storing the information w h i c h passes between programmer a n d operating system, e.g. the operating system generates certain error codes store in the specific address in this area during the execution of user program which can be read by user program; or the user may also give additional information to the operating system before execution of user program by writing some codes in the specific address in this area, etc.

c. I/O Status Memory - I/O Image Table

Another portion of RAM is allocated for the storage of current I/O status. Every single input/output module has been assigned to it a particular location within the input/output image table. The location within the input and output image tables are identified by addresses, each location has its own unique address.

During the execution of user program, the microprocessor scans the user program and interpret the user commands, the status of input modules used are read from the input image table (not directly from the input module itself). Various output device status generated during the execution of user program are stored in the output image table (not directly to output modules). (*Find out about input scan and output scan*.)

d. Data Memory

Whenever timers, counters, mathematics and process parameters are required, an area of memory must be set aside for data storage. The data storage portion of memory is allocated for the storage of such items as timers or counter preset/accumulated values, mathematics instruction data and results, and other miscellaneous data and information which will be used by any data manipulation functions in the user program.

Some manufacturers subdivide the data memory area into two sub-memories, one for fixed data and other for variable data. The fixed data portion can only be programmed via the programming device. The CPU is not permitted to place data values in this area. The variable portion of the data memory is available to the CPU for data storage.

e. User Program Memory

The final area of memory in a PLC is allocated to the storage of the user program. It is this memory area that the executive program instructs the microprocessor to examine or 'scan' to find the user instructions. The user program area may be subdivided if the CPU allocates a portion of this memory area for the storage of ASCII messages, subroutine programs, or other special programming functions or routines. In the majority PLCs, the internal data storage and user program areas are located in RAM.

Several systems do offer an option that places both the user program and the fixed data storage areas in EPROM type memory. The user can develop program in RAM and run the system to ensure correct operation. Once the user is satisfied that the programming is correct, a set of EPROMs is then duplicated from the RAM. Then the user can shut down the CPU and replaces the RAM with the newly programmed EPROM. Any future change would require that the EPROMs be reprogrammed.

Discrete I/O

A “discrete” data point is one with only two states on and off. Process switches, pushbutton switches, limit switches, and proximity switches are all examples of discrete sensing devices. In order for a PLC to be aware of a discrete sensor’s state, it must receive a signal from the sensor through a discrete input channel.

Inside the discrete input module is (typically) a light-emitting diode (LED) which will be energized when the corresponding sensing device turns on. Light from this LED shines on a photo-sensitive device such as a phototransistor inside the module, which in turn activates a bit (a single element of digital data) inside the PLC's memory. This opto-coupled arrangement makes each input channel of a PLC rather rugged, capable of isolating the sensitive computer circuitry of the PLC from transient voltage "spikes" and other electrical phenomena capable of causing damage.

Indicator lamps, solenoid valves, and motor contactors (starters) are all examples of discrete control devices. In a manner similar to discrete inputs, a PLC connects to any number of different discrete final control devices through a discrete output channel. Discrete output modules typically use the same form of opto-isolation to allow the PLC's computer circuitry to send electrical power to loads: the internal PLC circuitry driving an LED which then activates some form of photosensitive switching device. Alternatively, small electro mechanical relays may be used to interface the PLC's output bits to real-world electrical control devices.

Analog I/O

In the early days of programmable logic controllers, processor speed and memory were too limited to support anything but discrete (on/off) control functions. Consequently, the only I/O capability found on early PLCs were discrete in nature2. Modern PLC technology, though, is powerful enough to support the measurement, processing, and output of analog (continuously variable) signals.

All PLCs are digital devices at heart. Thus, in order to interface with an analog sensor or control device, some "translation" is necessary between the analog and digital worlds. Inside every analog input module is an ADC, or Analog-to-Digital Converter, circuit designed to convert an analog electrical signal into a multi-bit binary word. Conversely, every analog output module contains a DAC, or Digital-to-Analog Converter, circuit to convert the PLC's digital command words into analog electrical quantities.

Analog I/O is commonly available for modular PLCs for many different analog signal types, including:

1. Voltage (0 to 10 volt, 0 to 5 volt) 2. Current (0 to 20 mA, 4 to 20 mA)
3. Thermocouple (millivoltage) 4. RTD (millivoltage) 5. Strain gauge (millivoltage)

Programming of PLC:

The PLC can be programmed in different ways. There are various methods for entering and interconnecting various logic elements. Some of these can be explained as follows:

1. Entry of ladder logic diagram 2. Low level computer languages
3. High level computer languages 4. Functional blocks 5. Sequential function chart

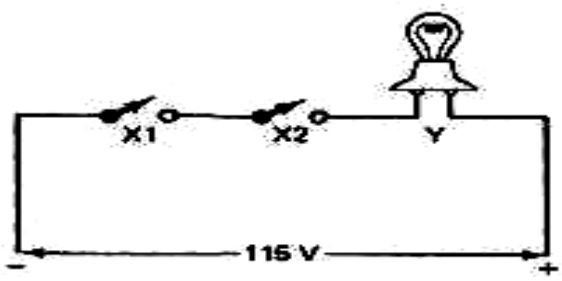
Programming Devices are:

Programming Console, Hand Programmer, PC

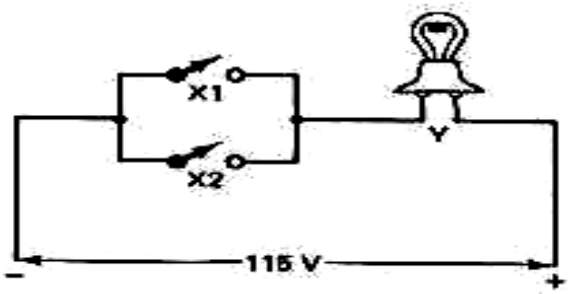
Ladder Logic Diagram

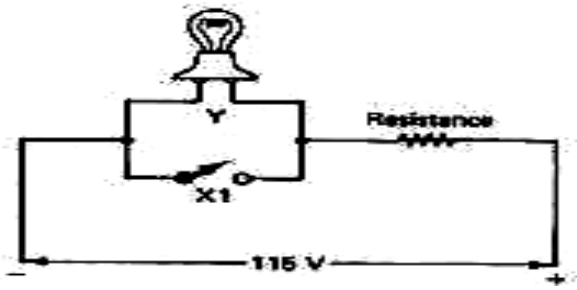
The ladder diagrams are based on three logic controls: 1. AND 2. OR 3. NOT

1. Logic AND operation



2. Logic OR operation





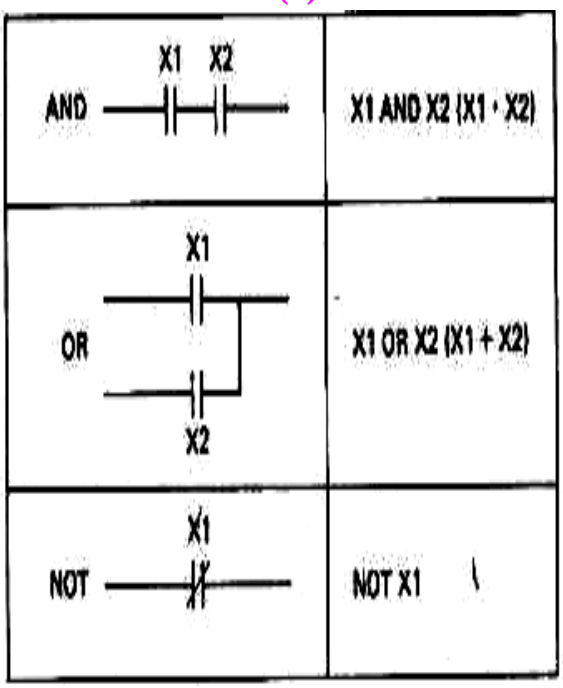
3. Logic NOT operation

(i)

Ladder symbol	Typical hardware components
(a)	Normally open contacts (switch, relay, other ON/OFF devices)
(b)	Normally closed contacts (switch, relay, etc.)
(c)	Output loads (motor, lamp solenoid, alarm, etc.)
(d) TMR 3s	Timer
(e) CTR	Counter

(i). Ladder logic symbols

(ii)



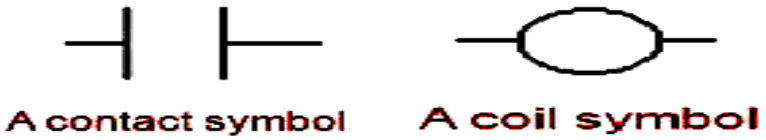
(ii). Implementing the switching operations using ladder symbol

Ladder Diagram:

- **First Step** : Translate all of the items we're using into symbols the PLC understands.
- **Second step** : We must tell the PLC where everything is located. In other words we have to give all the devices an address.
- **Final step** : We have to convert the schematic into a logical sequence of events.

First Step:

- The PLC doesn't understand terms like switch, relay, bell, etc.
- It prefers input, output, coil, contact, etc.
- It doesn't care what the actual input or output device actually is. It only cares that its an input or an output.
- First we replace the battery with a symbol. This symbol is common to all ladder diagrams. We draw what are called **bus bars**.
- These simply look like two vertical bars. One on each side of the diagram. Think of the left one as being + voltage and the right one as being ground. Further think of the current (logic) flow as being from left to right.
- Next we give the **inputs** a symbol. In this basic example we have one real world input. (i.e. the switch).
- We give the input that the switch will be connected to the symbol shown below. This symbol can also be used as the **contact of a relay**.



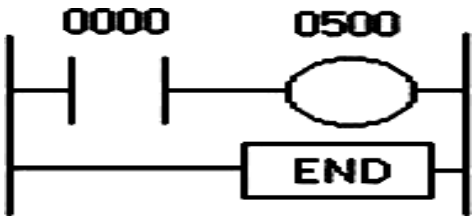
- Next we give the **outputs** a symbol. In this example we use one output (i.e. the bell).
- We give the output that the bell will be physically connected to the symbol shown below. This symbol is used as the coil of a relay.
- The AC supply is an external supply so we don't put it in our ladder. The PLC only cares about which output it turns on and not what's physically connected to it.

Second Step:

- We must tell the PLC where everything is located. In other words we have to give all the devices an address.
- Where is the switch going to be physically connected to the PLC? How about the bell? We start with a blank road map in the PLCs town and give each item an address.
- Could you find your friends if you didn't know their address? You know they live in the same town but which house? The PLC town has a lot of houses (inputs and outputs) but we have to figure out who lives where (what device is connected where).
- We'll get further into the addressing scheme later. The PLC manufacturers each do it a different way! For now let's say that our input will be called "0000". The output will be called "500".

Final Step:

- Convert the schematic into a logical sequence of events.
- The program we're going to write tells the PLC what to do when certain events take place.
- In our example we have to tell the PLC what to do when the operator turns on the switch.
- Final converted diagram.
- We eliminated the real world relay from needing a symbol as shown below.



A LoaD (contact) symbol

Basic Instructions

Load:

- The load (LD) instruction is a **normally open contact**. It is sometimes also called examine if on (**XIO**) (as in examine the input to see if its physically on). The symbol for a load instruction is shown above.
- This is used when an input signal is needed to be present for the symbol to turn on.
- When the physical input is on we can say that the instruction is True.
- We examine the input for an on signal. If the input is physically on then the symbol is on.
- An on condition is also referred to as a logic 1 state.

Load Bar:

- The Load bar instruction is a **normally closed contact**. It is sometimes also called LoaDNot or examine if closed (**XIC**) (as in examine the input to see if its physically closed) The symbol for a load bar instruction is shown below.

A LoaDNot (normally closed contact) symbol

Physical State	Instruction	Logic
OFF	TRUE	0
ON	FALSE	1

- This is used when an input signal does not need to be present for the symbol to turn on.
- When the **physical input is off** we can say that the **instruction is True**.
- We examine the input for an off signal. If the input is physically off then the symbol is on.
- With most PLCs this instruction (**Load** or **Load bar**) MUST be the first symbol on the left of the ladder.

Out:

- The Out instruction is sometimes also called an **Output Energize instruction**. The output instruction is like a **relay coil**. Its symbol looks as shown below.

An OUT (coil) symbol

An OUTBar (normally closed coil) symbol

- When there is a path of True instructions preceding this on the ladder rung, it will also be True.
- When the **instruction is True it is physically ON**.
- We can think of this instruction as a normally open output.

Out Bar:

- The Outbar instruction is sometimes also called an OutNot instruction.
- The Outbar instruction is like a **normally closed relay coil**. Its symbol looks like that shown below.

Among these, ladder logic diagram is most frequently used as whenever a personnel wants to change the PLC, he does not have to learn an entirely new programming language. Only the knowledge of the circuit diagram is enough.

This method includes the direct entry of the logic diagram into the PLC memory. This method requires the use of a keyboard and a display screen with graphics capability to display the symbols of the components and their inter relationships in the ladder logic diagram. Programming is accomplished by inserting appropriate components in the rungs of the ladder diagram.

- Ladder logic diagram uses graphic symbols similar to relay schematic circuit diagrams.
- It consists of two vertical lines representing the **power rails**
- Circuits are connected as horizontal lines between these two vertical lines. Such horizontal lines are called as **Rungs.**
- Each rung contains atleast one input and one output.
- A particular input or output can appear in more than one rung of ladder.
- Output is connected at right side and input is connected and input is connected at left side (for each rung)
- Output (which is on right side) cannot be directly connected with left side
- The ladder logic diagram consists of two types of components: Contacts & Coils

The various devices that are based on the binary logic can be stated as follows:

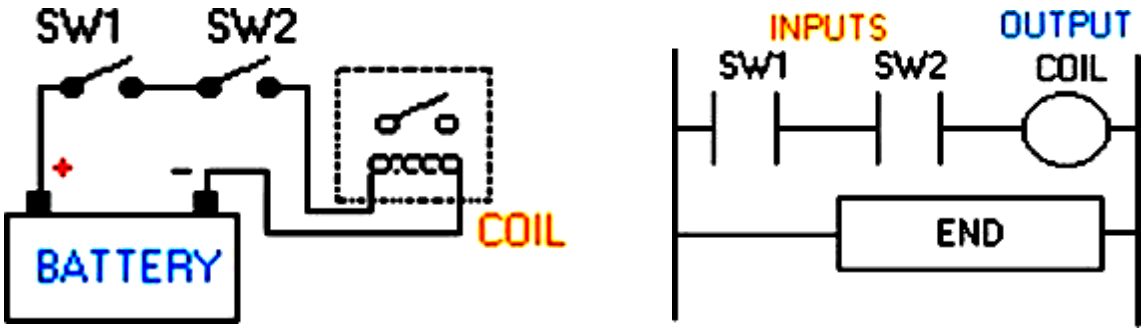
DEVICE	ONE / ZERO
INPUT	
Limit switch	Contact / no contact
Photo detector	Contact / no contact
Pushbutton switch	On /off
Timer	On / off
Control relay	Contact / no contact
Circuit breaker	Contact / no contact
OUTPUT	
Motor	On / off
Alarm buzzer	On / off
Control relay	Contact / no contact
Lights	On / off
Valves	Closed / open
Clutch	Engaged / not engaged
Solenoid	Energized / not energized

- Contacts are used to represent loads such as motors, relays, solenoids, timers, counters, etc.
- The program is entered rung by rung in the logic diagram.

- **Disadvantage:** Ladder logic diagrams are based on the ON-OFF operation. They are entirely based on the logic level 1 and logic level 0. This can be a disadvantage in using the ladder logic programming.

A Simple Example 1:

- In the above circuit, the coil will be energized when there is a closed loop between the + and - terminals of the battery.
- We can simulate this same circuit with a ladder diagram



- A ladder diagram consists of individual rungs just like on a real ladder.
- Each rung must contain one or more inputs and one or more outputs.
- The first instruction on a rung must always be an input instruction and the last instruction on a rung should always be an output (or its equivalent).
- Notice in this simple one rung ladder diagram we have recreated the external circuit above with a ladder diagram.
- Here we used the Load and Out instructions.
- Some manufacturers require that every ladder diagram include an END instruction on the last rung. Some PLCs also require an ENDH instruction on the rung after the END rung.

Example 2:

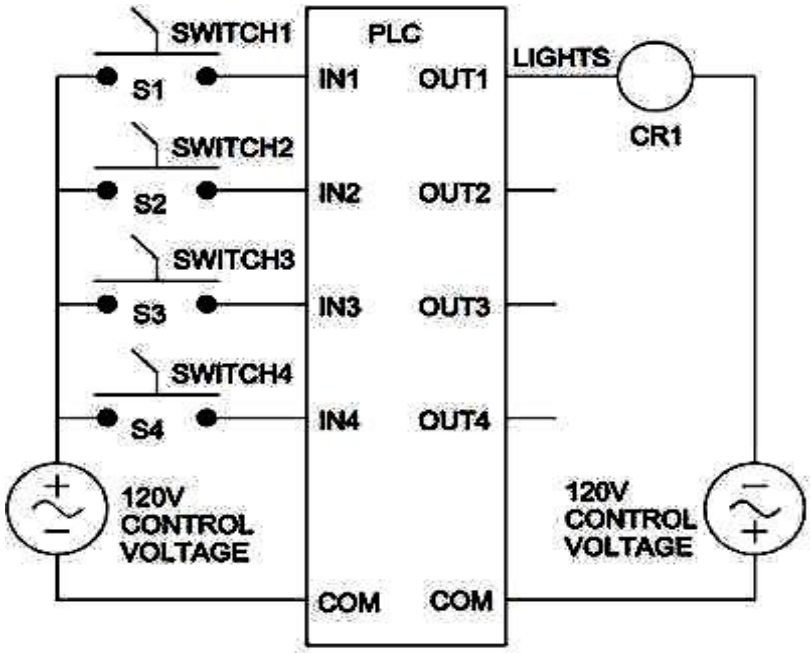
Lighting controlsystem:

A lighting control system is to be developed. The system will be controlled by four switches, SWITCH1, SWITCH2, SWITCH3, and SWITCH4. These switches will control the lighting in a room based on the following criteria:

1. Any of three of the switches SWITCH1, SWITCH2, and SWITCH3, if turned ON can turn the lighting on, but all three switches must be OFF before the lighting will turn OFF.
2. The fourth switch SWITCH4 is a Master Control Switch. If this switch is in the ON position, the lights will be OFF and none of the other three switches have any control.

The first item we may accomplish is the drawing of the **controller wiring diagram**. All we need do is connect all switches to inputs and the lighting to an output and note the numbers of the inputs and output associated with these connections.

The remainder of the task becomes developing the **ladder diagram**. The wiring diagram is shown in Figure.



Notice that all four switches are shown as **normally open selector switches** and the output is connected to a relay coil **CR1**. We are using the relay CR1 to operate the lights because generally the current required to operate a bank of room lights is higher than the maximum current a PLC output can carry.

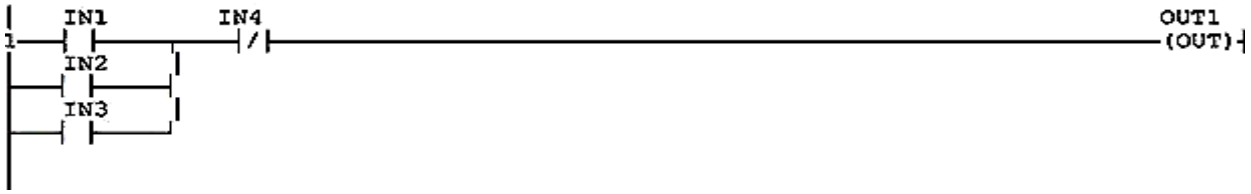
For this wiring configuration, the following definition list is apparent:

- **INPUT IN1** = SWITCH1
- **INPUT IN2** = SWITCH2
- **INPUT IN3** = SWITCH3
- **INPUT IN4** = SWITCH4 (Master Control Switch)
- **OUTPUT OUT1** = Lights control relay coil CR1

This program requires that when SWITCH4 is ON, the lights must be OFF. In order to do this, it would appear that we need a N/C SWITCH4, not a N/O as we have in our wiring diagram. However, keep in mind that once an input signal is brought into a PLC, we may use as many contacts of the input as we need in our program, **and the contacts may be either N/O or N/C**.

Therefore, we may use a N/O switch for SWITCH4 and then in the program, we will logically invert it by using N/C IN4 contacts.

The ladder diagram to implement this example problem is shown in Figure.



The ladder was printed using graphics characters (extended ASCII characters).

Notice the normally closed contact for IN4. A normally closed contact represents an inversion of the assigned element, in this case IN4, which is defined as SWITCH 4. Remember, SWITCH 4 has to be in the OFF position before any of the other switches can take control. In the OFF position, SWITCH 4 is open.

This means that IN4 will be OFF (de -energize d). So, in order for an element assigned to IN4 to be closed with the switch in the OFF position, it must be shown as **a normally closed contact**. When SWITCH 4 is turned ON, the input, IN4, will become active (energized). If IN4 is ON, a normally closed IN4 contact will open.

With this contact open in the ladder diagram, none of the other switches will be able to control the output.

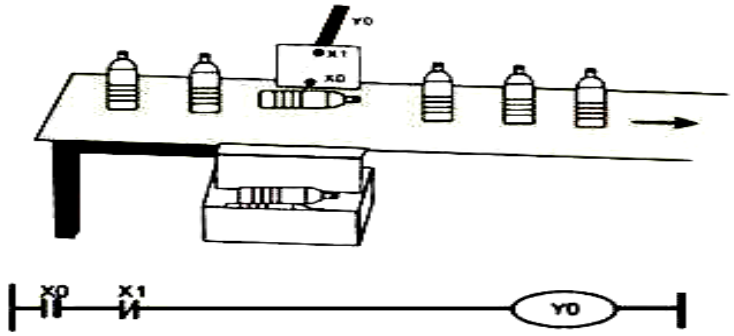
REMEMBER: A normally closed switch will open when energized and will close when de-energized.

Example 3:

Devices:

Device	Function
X0	X0=ON when the detected input signal from the bottle bottom is sheltered
X1	X1=ON when the detected input signal from the bottle neck is sheltered
Y 0	Pneumatic pushing pole

Detecting the standing bottles on the conveyor and pushing the fallen bottles out



Program description

- If the bottle on the conveyor belt is upstanding, the input signal from the monitoring photocell at both bottle bottom and bottle neck will be detected. In this case, X0=ON and X1=ON. The normally open (NO) contact X0 will be activated as w ell as the normally closed (NC) contact X1. Y 0 remains OFF and pneumatic pushing pole will not perform any action.

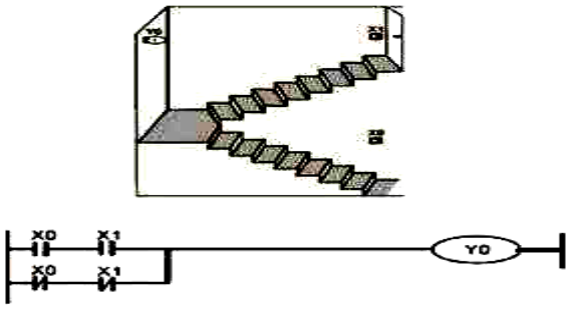
- If the bottle from the conveyor belt is down, only the input signal from monitoring photocell at the bottle bottom will be detected. I n this case, X0=ON, X1=O FF. The state of the output Y 0 will be ON because the NO contact X activates and the NC contact X1 remains OFF. The pneumatic pushing pole will push the fallen bottle out of the conveyor belt

Example 4:

Devices:

Device	Function
X0	X0 turns ON when the bottom switch is turned to the right
X1	X1 turns ON when the top switch is turned to the right
Y 1	Stair light

Setting up a lightning system for users to switch on/off the lights whether they are bottom or the top of the stairs



Program Description

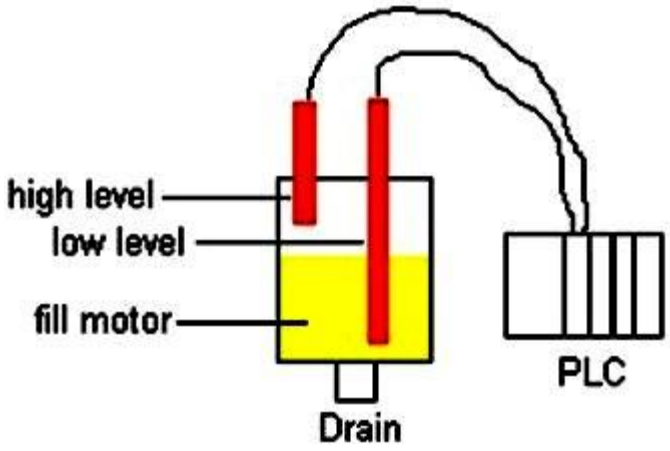
- If the states of the bottom switch and the top switch are the same, both ON or OFF, the light will be ON. If different, one is ON and the other is OFF, the light will be OFF.
- When the light is OFF, users can turn on the light by changing the state of either top switch or the bottom switch of the stairs. Likewise, when the light is ON, users can turn off the light by changing the state of one of the two switches.

Example 5:

A Level Application:

- We are controlling lubricating oil being dispensed from a tank.
- This is possible by using two sensors.
- We put one near the bottom and one near the top, as shown in the picture.

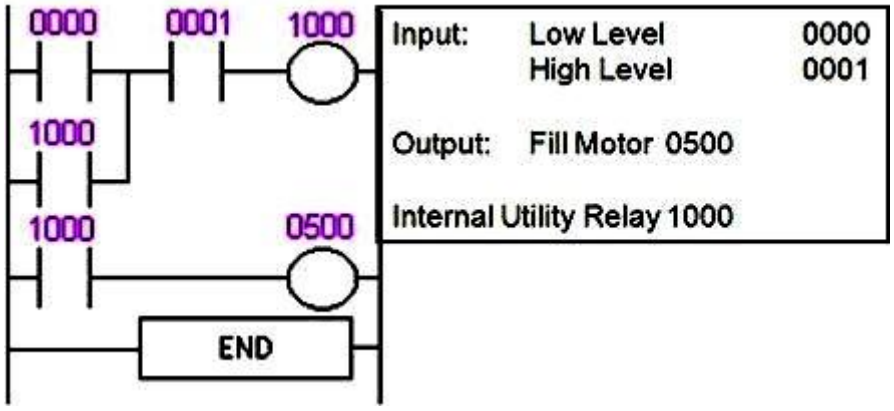
Here, we want the fill motor to pump lubricating oil into the tank until the high level sensor turns on. At that point we want to turn off the motor until the level falls below the low level sensor. Then we should turn on the fill motor and repeat the process.



Dispensing oil from a tank

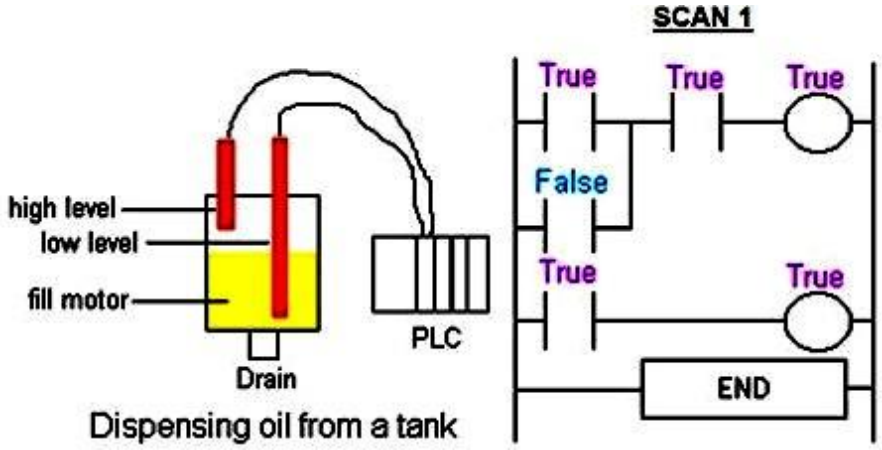
Here we have a need for 3 I/O (i.e. Inputs/Outputs):

- 2 are inputs (the sensors) and 1 is an output (the fill motor).
- Both of our inputs will be NC (normally closed) fiber-optic level sensors. When they are NOT immersed in liquid they will be ON. When they are immersed in liquid they will be OFF.



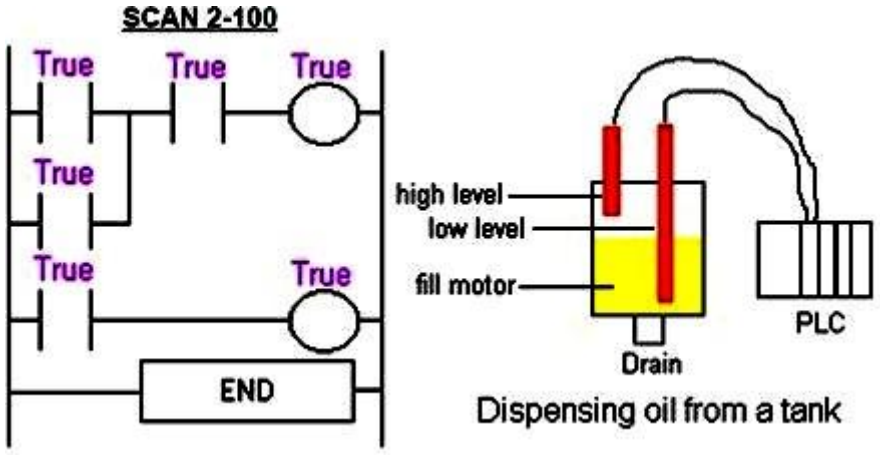
Program Scan:

- Initially the tank is empty. Therefore, input 0000 is TRUE and input 0001 is also TRUE.

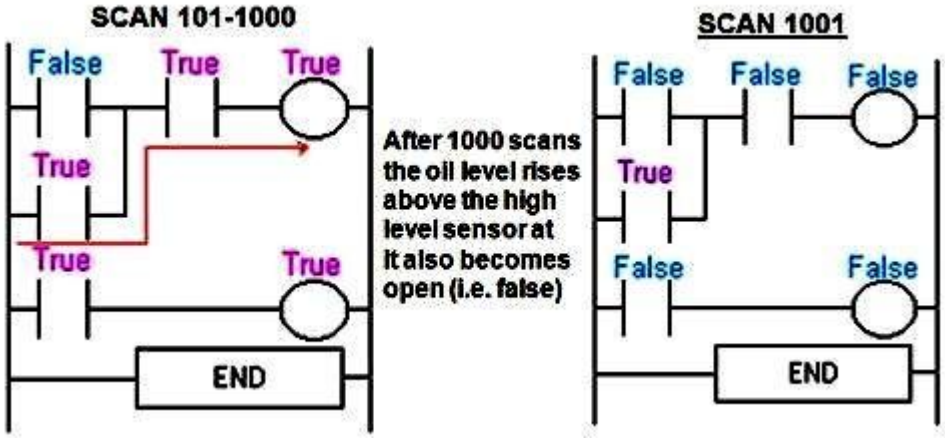


Dispensing oil from a tank

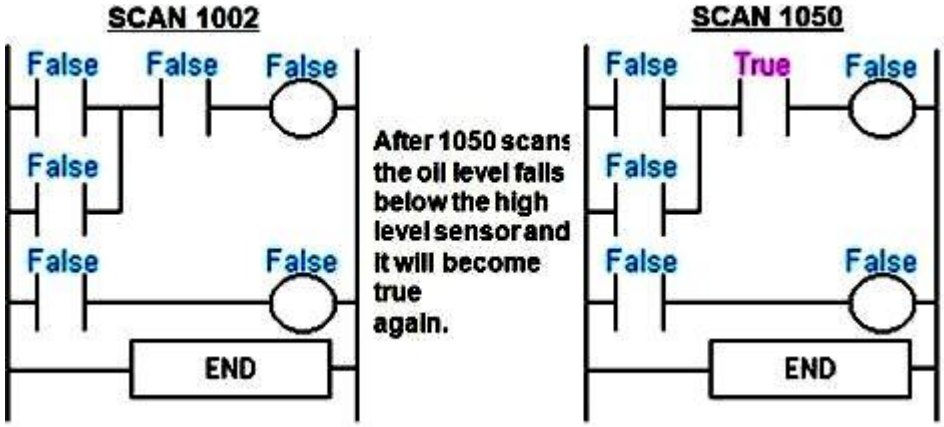
- Gradually the tank fills because 500(fill motor) is on.
- After 100 scans the oil level rises above the low level sensor and it becomes open. (i.e. FALSE).



- Even when the low level sensor is false there is still a path of true logic from left to right. This is why we used an internal relay. Relay 1000 is latching the output (500) on. It will stay this way until there is no true logic path from left to right.(i.e. when 0001 becomes false).

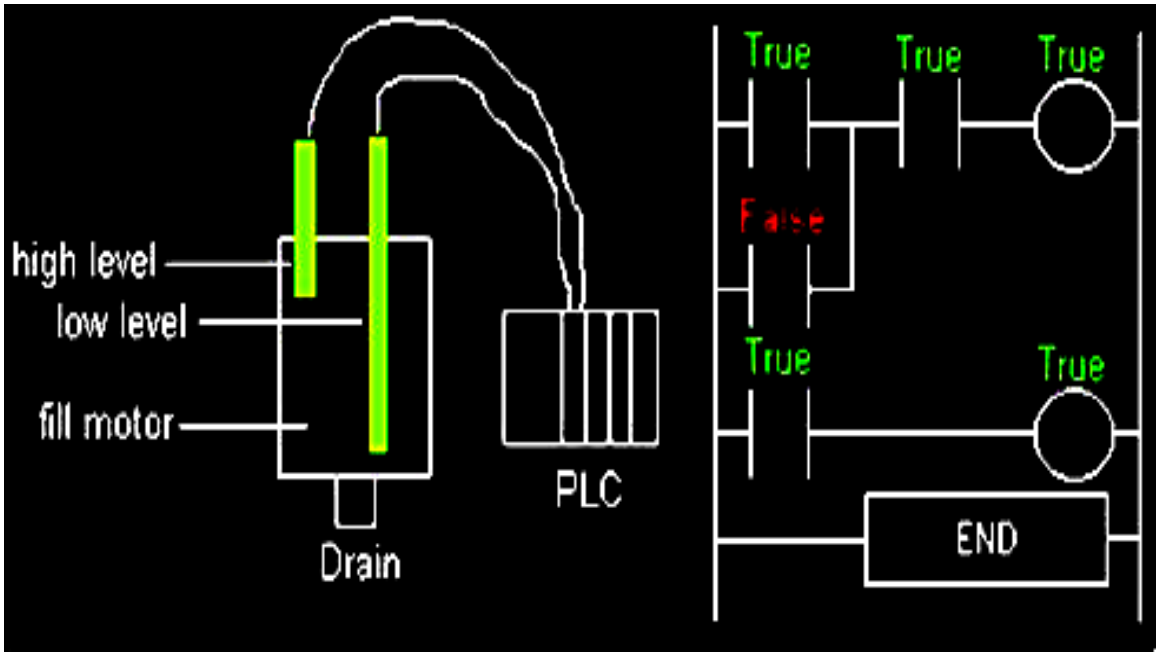


- Since there is no more true logic path, output 500 is no longer energized (true) and therefore the motor turns off.



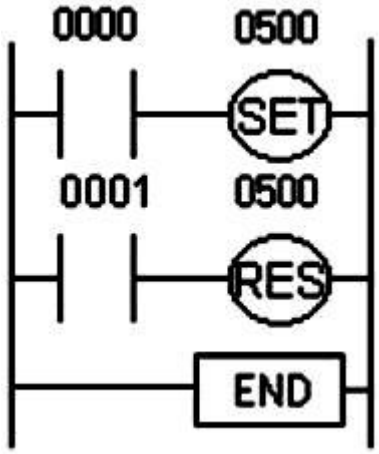
- Even though the high level sensor became true there still is NO continuous true logic path and therefore coil 1000 remains false!.

- After 2000 scans the oil level falls below the low level sensor and it will also become true again.
- At this point the logic will appear the same as SCAN 1 above and the logic will repeat as illustrated above.

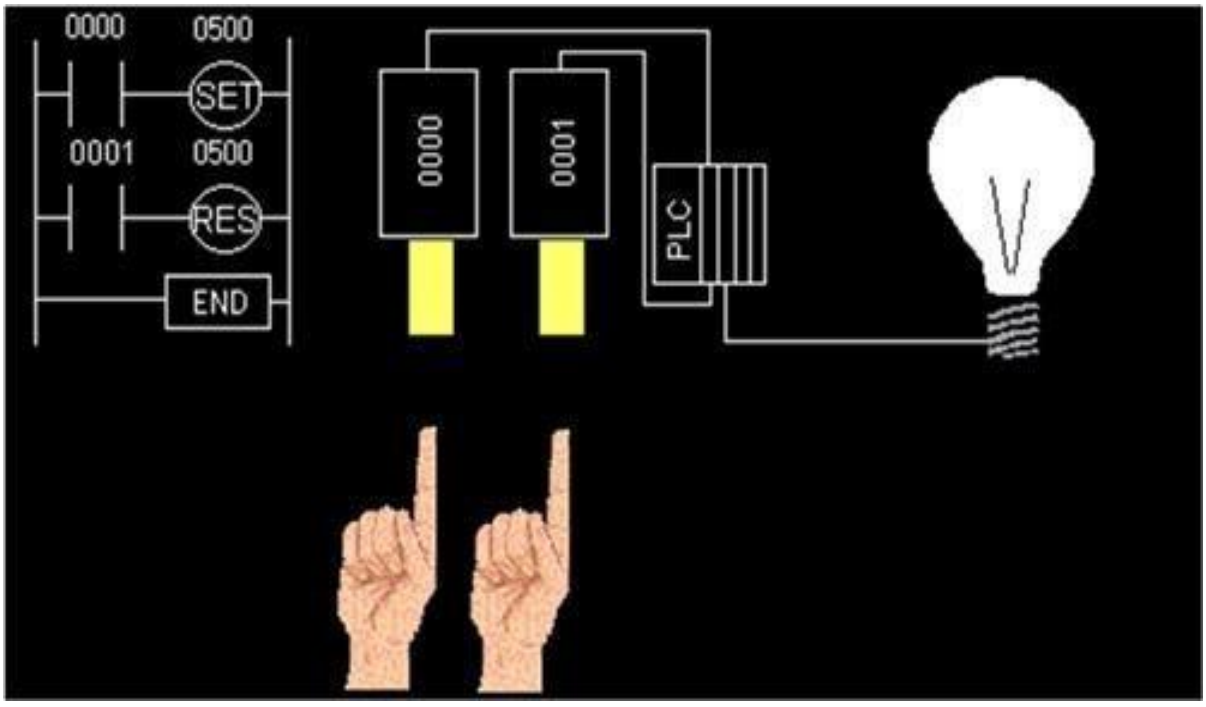


Latch Instruction:

- The latching instructions let us use momentary switches and program the plc so that when we push one the output turns on and when we push another the output turns off.
- Picture the remote control for your TV. It has a button for ON and another for OFF:
- When I push the ON button the TV turns on.
- When I push the OFF button the TV turns off.
- I don't have to keep pushing the ON button to keep the TV on. This would be the function of a latching instruction.
- The latch instruction is often called a SET or OTL (output latch).
- The unlatch instruction is often called a RES (reset), OUT (output unlatch) or RST (reset). The diagram below shows how to use them in a program.



Here we are using 2 momentary push button switches. One is physically connected to input 0000 while the other is physically connected to input 0001. When the operator pushes switch 0000 the instruction "set 0500" will become true and output 0500 physically turns on. Even after the operator stops pushing the switch, the output (0500) will remain on. It is latched on. The only way to turn off output 0500 is turn on input 0001. This will cause the instruction "res 0500" to become true thereby unlatching or resetting output 0500.



• What would happen if input 0000 and 0001 both turn on at the exact same time
• Will output 0500 be latched or unlatched?
• To answer this question we have to think about the scanning sequence. The ladder is always scanned from top to bottom, left to right.
• The first thing in the scan is to physically look at the inputs.
• 0000 and 0001 are both physically on.
• Next the PLC executes the program.
• Starting from the top left, input 0000 is true therefore it should set 0500.
• Next it goes to the next rung and since input 0001 is true it should reset 0500.
• The last thing it said was to reset 0500. Therefore on the last part of the scan when it updates the outputs it will keep 0500 off. (i.e. reset 0500).

PLC features and benefits

Solid-state Components	- Low component failure - Low space and power consumption - No mechanical wear
Microprocessor-based	- An intelligent decision-making device - Communication with other smart devices - Multifunctional capabilities
Programmable	- Simplifies logic changes - Allows flexible control system design - Allows better accuracy and repeatability
Modular Components	- Easily installed and replaced - Minimizes hardware purchases - Flexible installation configurations - Neat appearance inside control panel - Easily wired and maintained
Diagnostic Indicators	- Reduces troubleshooting/downtime - Helps isolate field wiring problems - Signal correct operation and faults
Variety of Standard I/O Interfaces Modules	- Controls variety of standard devices - Eliminates customized interfaces - Reduces engineering design time/costs - Allows standardized wiring diagrams
Quick I/O Disconnects	- Serviceable without disturbing field wiring
Local and Remote Input/Output Subsystems	- I/O interfaces can be conveniently placed - Elimination of long wire/conduit runs
Software Timers & Counters	- Eliminates hardware cost and problems - Achieves better timing accuracy - Easily changed presets
Software Control Relays	- Eliminates hardware cost and problems - Reduces space requirements - Unlimited number of relay contacts
All Process Data Stored in Memory	- Production management and maintenance information can be used to generate reports

Part- A

1) Draw and explain block diagram of PLC.
2) What is ladder diagram? Explain with suitable example.
3) Write a short note of different type of register in PLC.
4) Describe application of PLC in power system.
5) Describe application of PLC in control drives.
6) Write a short note on timer and counter functions with reference of PLC.
7) Write a short note on history of PLC
8) Explain number comparison functions of PLC.
9) Explain skip and master control relay functions with reference of PLC.
10) Write advantages and different application of PLC

Part- B

1) Write a short note on PLC matrix functions.
2) Write a short note on sequencer function.
3) Explain on off mechanism operation in PLC.
4) What is hot rail and cold rail in terms of PLC.
5) Draw and explain input and out mechanism in PLC
6) Draw and explain block diagram of PLC

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT IV – LARGE SCALE CONTROL SYSTEM– SEIA 1503

UNIT 3 DISTRIBUTED CONTROL SYSTEMS (DCS)

Evolution of DCS - building blocks - different architectures-comparison of architectures-detailed descriptions and functions of local control units - basic elements & functions-operator stations - data highways - redundancy concepts.

3.1 Distributed control system (DCS)

DCS is a control system for a plant or process, where the control elements are distributed throughout the system. It is in contrast to non-distributed systems, that uses a single controller at a central location. In a DCS, a hierarchy of controllers are connected by communication networks for command and for monitoring. To command and to monitor a hierarchy of controllers is connected by communications networks. Example scenarios where DCS might be used are given below:

- Chemical plants
- Petrochemical plants (oil) and refineries
- Pulp and Paper Mills
- Boiler controls and power plant systems
- Nuclear power plants
- Environmental control systems
- Water management systems
- Metallurgical process plants
- Pharmaceutical manufacturing plants
- Sugar refining plants
- Dry cargo and bulk oil carrier ships
- Formation control of multi-agent systems

The processor receives information from input modules and sends information to output modules. The input modules receive information from input instruments in the process and transmit instructions to the output instruments in the field. Computer buses or electrical buses connect the processor and modules through multiplexer or demultiplexers. Also it connect the distributed controllers with the central controller and finally to the Human Machine Interface (HMI) or control consoles. Early minicomputers were used in the control of industrial processes ever since the beginning of the 1960s. For example IBM 1800, was an early computer that had input/output hardware to gather process signals in a plant for conversion from field contact levels (for digital points) and analog signals to the digital domain.

The first industrial control computer system was built in the year 1959 at the Texaco Port Arthur, Texas, refinery with an RW-300 of the Ramo-Wooldridge Company. DCS was introduced in 1975, where both Honeywell and Japanese electrical engineering firm Yokogawa introduced their own independently produced DCSs at roughly the same time, with the TDC 2000 and CENTUM systems. Also US -based Bristol introduced their UCS 3000 universal controller in 1975. In 1980, Bailey (now part of ABB) introduced the NETWORK 90 system. In 1980, Fischer & Porter Company (now also part of ABB) introduced DCI-4000 (DCI stands for Distributed Control Instrumentation). DCS largely came due to the increased availability of microcomputers and the proliferation of

microprocessors in the world of process control. Computers are applied to process automation for some time in the form of both Direct Digital Control (DDC) and Set Point Control. In the early 1970s Taylor Instrument Company, (now part of ABB) developed the 1010 system, Foxboro the FOX1 system and Bailey Controls the 1055 systems. All of them were DDC applications implemented within minicomputers (DEC PDP-11, Varian Data Machines, MODCOMP etc.) and connected to proprietary Input / Output hardware. Central to the DCS model was the addition of control function blocks. One of the first embodiments of object-oriented software, function blocks were self contained "blocks" of code that emulated analog hardware control components and performed tasks that are essential to process control, such as execution of PID algorithms. Function blocks continue to endure as the predominant method of control for DCS suppliers, and are supported by key technologies such as Foundation Field bus today.

Digital communication between distributed controllers, workstations and other computing elements (peer to peer access) are one of the primary advantages of DCS. Attention was duly focused on the networks, that provides all important lines of communication that is , for process applications, specific functions such as determinism and redundancy. Hence many suppliers embraced the IEEE 802.4 networking standard. This decision set the stage for the wave of migrations essential when information technology moved into process automation and IEEE 802.3 rather than IEEE 802.4 prevailed as the control LAN.

DCS conveyed distributed intelligence to the plant and built up the presence of computers and microprocessors in process control, it didn't manage the cost of the scope and openness important to bind together plant asset necessities. In various cases, the DCS was just a digital replacement of the same functionality provided by analog controllers and a panel board display. In 1980s, users began to look at DCSs as a development of the basic process control.

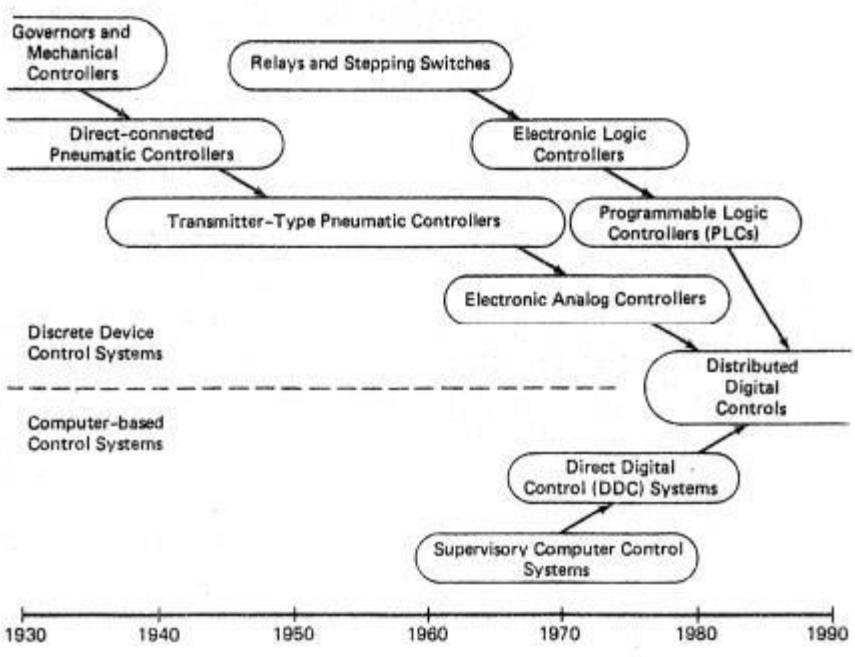


Fig. 3.1 Evolution of DCS

3.2 Traditional Control System Developments

The idea of distributed control systems is not a new one. The early discrete device control systems listed in the figure above were distributed around the plant. Individual control devices such as governors and mechanical controllers were situated at the process equipment to be control local readouts of set points. Control outputs were also available, and a means to change the control mode from manual to automatic are also usually provided. It is up to the operator to coordinate the control of the numerous gadgets that made up the total process. These controllers provides a more flexible way to select and adjust the control algorithms, yet all of the elements of the control loop such as sensor, controller, operator interface, and output actuator were still located in the field. There was no mechanism for communication between controllers.
Later in 1930's a new architecture was developed in which measurements made at the process were converted to pneumatic signals at standard levels, which were then transmitted to the central station. The required control signals were computed at this location, then transmitted back to the actuating devices. The great advantage of this architecture is that all the necessary information will be available to the operator at the central location. Thus, the operator was able to make better control decisions to operate the plant with a greater degree of safety and economic return.

Later in 1950s and early 1960s, the technology started to shift from pneumatics to electronics . This change reduced the installation cost and also eliminated the time lag in pneumatic systems . These advantages became more significant as the size of the plants increased. Another consequence of the centralized control architecture was the development of the split controller structure . In this type of controller, the operator display section of the controller is panel mounted in the control room and the computing section will be located in a separate room.

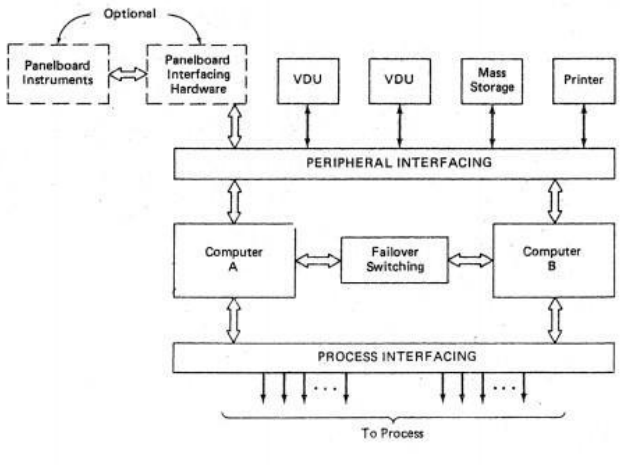
3.3 Resulting System Architectures

As a result of the developments , two industrial control system architectures came into existence by the end of the 1970s . The first architecture is a hybrid one, which makes use of a combination of discrete control hardware and computer hardware in a central location to implement the necessary control functions. In hybrid architecture, the local control of the plant operations is implemented by using discrete analog and sequential logic controllers . Operator interface is usually provided by panel board instruments. The plant management operations are performed by supervisory computer and its associated data acquisition system. The supervisory computer is used to perform operations such as operating point optimization, alarming, data logging, and historical data storage and retrieval. It is also used to operating point optimization, alarming, data logging, and historical data storage and retrieval.

In the second architecture, all the system functions are implemented in high performance computer hardware in a central location. To accomplish this task, redundant computers are required so that the failure of a single computer does not shut the whole process .Operator interface for the plant management functions is provided using

computer-driven VDUs. Operator interface for continuous and sequential closed-loop control are also implemented using VDUs.

The computers can be interfaced to standard panel board instrumentation so that the operator in charge can use a more familiar set of operator interface. The major difference between the two architecture is the location and the implementation of the first- level continuous and sequential logic control functions . By the late 1970s, the hybrid system architecture became a more popular approach in industrial practice . The chemical and petroleum process industries heavily favored this approach.



Central Computer System Architecture.

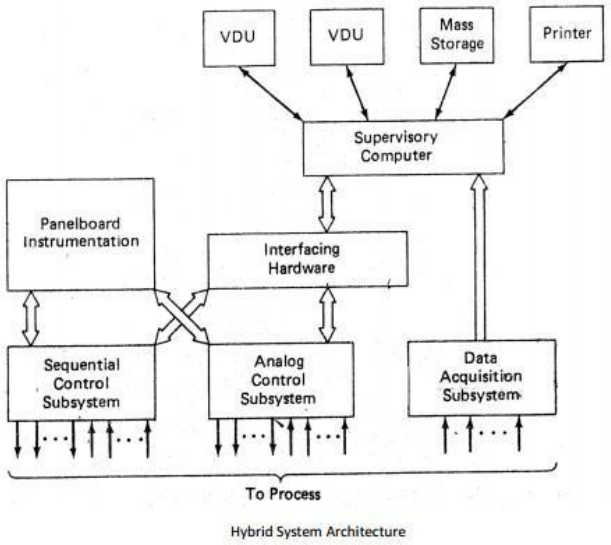


Fig.3.2 Hybrid system Architecture

Fig.3.3 Central computer system Architecture

Though centralized computer and hybrid system architectures provides significant advantages over existing architectures, they suffered from a number of disadvantages . They are- The CPU represents a single point failure that can shutdown the entire process if it is lost. To overcome the above drawback, another computer is used as a ‘‘hot standby’‘ to take over if the primary control computer fails. This approach lead to a new architecture which is more expensive than an analog control system that performs a comparable set of functions. Another problem with these computer-based systems has been that the software required to implement all these functions is extremely complex.

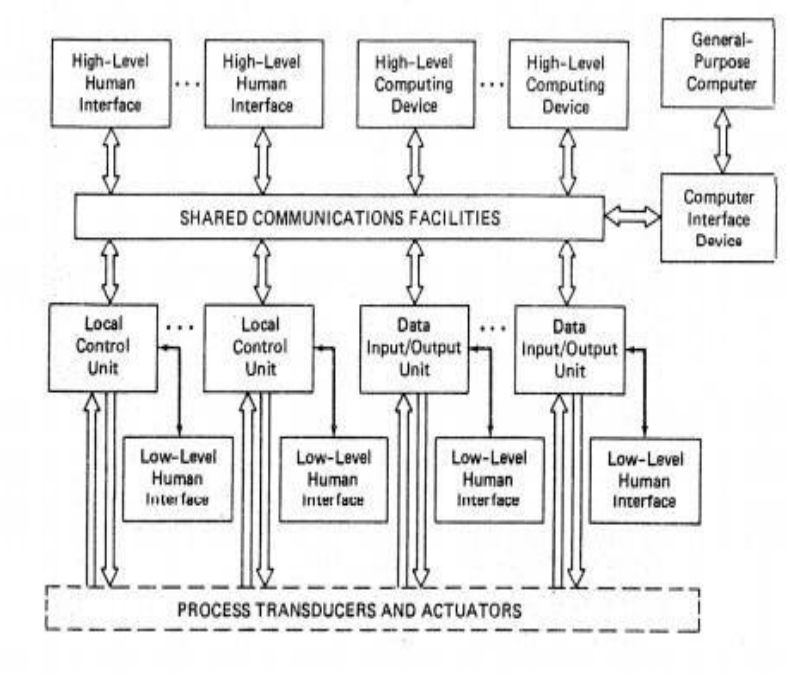


Fig.3.4 Generalized Distributed Control System Architecture

3.4 COMPARISON WITH PREVIOUS ARCHITECTURES

Scalability and expandability—The hybrid system architecture is quite modular than distributed system architecture. The central computer architecture is designed for only a small range of applications which is not cost-effective for applications smaller than its design size and it cannot be expanded once its memory and performance limits are reached.

Control capability—Hybrid architecture has only limited functions available in the hardware modules .To add a function involves adding hardware and rewiring the control system. While in central computer and distributed architectures advantages of using digital control is available

Operator interfacing capability—The operator interface in the hybrid system consists of conventional panel board instrumentation for control and monitoring functions and a separate video display unit. In the central computer and distributed architectures, VDUs

are used as the primary operator interface for supervisory control functions. The VDUs in the distributed system are driven by microprocessor which can be applied in a cost effective way to small systems as well as large ones.

Integration of system functions—A high degree of integration minimizes user problems in procuring, interfacing, starting up, and maintaining the system. The hybrid system is poorly integrated. The central computer architecture is well integrated because the functions are performed only by the same hardware. The distributed system lies somewhere in between, depending on how well the products are designed to work together.

Significance of single point fai1ure—In the central computer architecture, the failure of the supervisory computer will cause the entire plant to shut down unless a backup computer is used. Hence centralized architecture is very sensitive to single-point failures while the hybrid and distributed architectures are insensitive to single-point failures due to the modularity of their structure.

Installation costs—The installation costs of the hybrid system is high since custom wiring is needed for internal system interconnections and long wiring runs are needed from sensors to control cabinets and the large volume of control modules are required to implement the system.

The central computer architecture requires less cost the module inter- connection wiring are eliminated and VDUs are used to replace much of the panel-board instrumentation. The distributed system reduces costs further by using a communication system to replace the sensor wiring runs .

Maintainability—The hybrid system is particularly poor in this area because large number of spare modules are required. The central computer architecture is better than hybrid architecture since the range of module types is reduced . The maintainability of the distributed system architecture is excellent since there are only a few general-purpose control modules in the system. The spare parts and personnel training requirements are also minimal in distributed architecture.

3.5 A comparison of architectures

While the figure describes the basic elements of all microprocessor based local control units, the current offerings of controllers in the market place exhibit endless variations on this structure. The controllers differ in size, I/O capability, range of functions provided, and other architectural parameters depending on the application and the vendor who designed the equipment.

3.6 Architectural Parameters

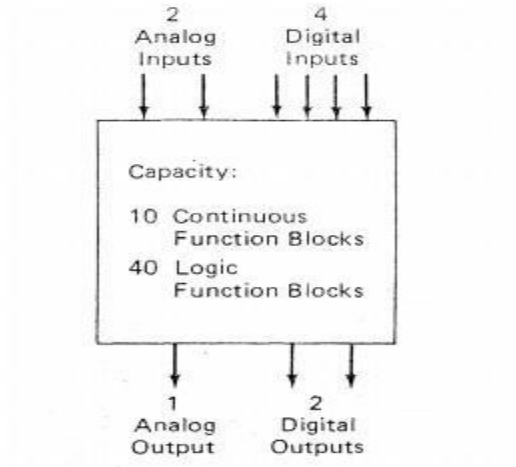
When evaluating the controllers in the market or when specifying a new one, the control system designer is facing with the problem of choosing a controller architecture that best meets the needs of the range of applications in which the controller is to be used. Few of the major architectural parameters that must be selected include the following:

1. Size of controller—This refers to the number of function blocks and/ or language statements that can be executed by the controller, and also the number of process I/O channels provided by the controller.
2. Functionality of controller—This refers to the mix of function blocks or language statements provided by the controller (e.g., continuous control, logic control,

arithmetic functions, or combinations of the above) .Also it refers to the mix of process input and output types (e.g, analog or digital) provided by the controller.

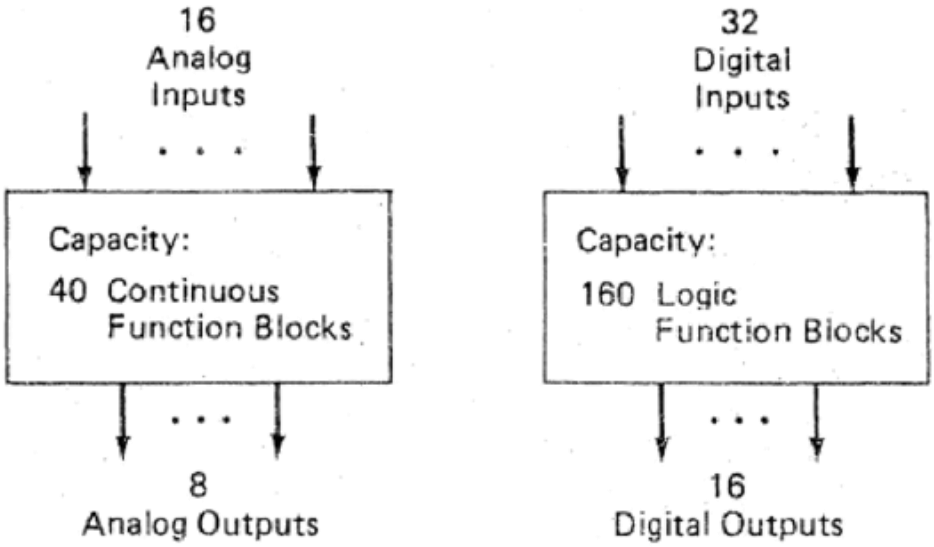
3. Performance of controller—It refers to the rate at which the controller scans inputs, processes function blocks or language statements, generates outputs, also includes the accuracy with which the controller performs these operations.
4. Communication channels out of controller— ln addition to process inputs and output channels, the controller must provide other communication channels to operator interface devices and to other controllers and devices in the system. The number, type and speed of these channels are key controller design parameters.
5. Controller output security—In a real-time process control system, a mechanism must be provided (usually manual backup or redundancy) to ensure that the control output is maintained despite a controller failure so that a process shutdown can be avoided.

Unfortunately, it is not generally possible to select any one of these architectural parameters independently from all of the others, since there is a great degree of interaction among them. Hence, selecting the best combination for the range of applications to be considered is more a matter of engineering judgment than a science. Each vendor of microprocessor based systems has a different view of the range of applications intended for the controller, and as a result designs an LCU architecture that quite often differs from that of its competitors. To illustrate some of the differences in LCU architectures, three representative LCU configurations are shown in Figures. They are not intended to represent particular commercially available products but, different classes of controllers on the market today.

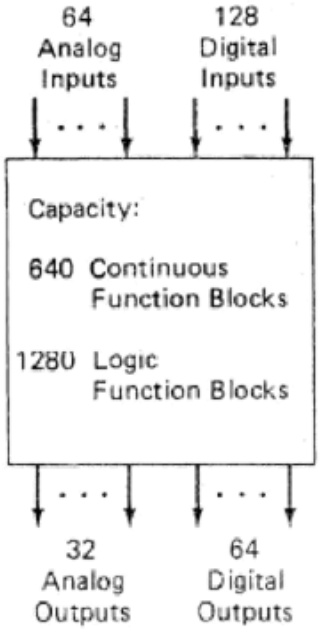


LCU Architecture—Configuration A

Fig.3.5 LCU Architecture – Configuration A



LCU Architecture—Configuration B



LCU Architecture—Configuration C

Fig.3.6 LCU Architecture – Configuration B and C

Configuration A represents a class of single-loop LCU that provides both analog and digital inputs and outputs and executes both continuous and logic function blocks. Configuration B represents an architecture in which two different types of LCUs

implement the full range of required continuous and logic functions. Configuration C gives a multiloop controller architecture in which both continuous and logic functions are performed.

Comparison of LCU Architectures

ARCHITECTURE PARAMETERS	CONFIGURATION A (SINGLE- LOOP)	CONFIGURATION B (2 LCU TYPES)	CONFIGURATION C (MULTI-LOOP)
Controller size	Number of functions needed for single PID loop or motor controller	Includes functions and I/O needed for eight control loops and a small logic controller	System size is equivalent to small DDC system
Controller Functionality	Uses both continuous and logic function blocks	Continuous and logic function blocks split between controllers	Uses both continuous and logic function blocks, can support high level languages
Controller Scalability	High degree of scalability from small to large systems	Requires both controller types even in small systems	Not scalable to very small systems
Controller Performance	Requirements can be met with inexpensive hardware	Because of functional split, performance requirements are not excessive	Hardware must be high performance to execute large number of functions
Communication Channels	Need inter module communications for control; only minimum needed for human interface	Functional separation requires close interface between controller types	Large communication requirement to human interface; minimal between controllers.
Controller output security	Controller has single loop integrity; usually only manual backup Is needed	Lack of single loop integrity requires redundancy in critical applications.	Size of controller requires redundancy in all applications

LCU Configuration A - In configuration A, the controller size is the minimum that required to perform a single loop of control or a single motor control function or other simple sequencing function. Two digital outputs are provided to allow the controller to drive a pulsed (raise/lower) positioner or actuator. Twice as many inputs as outputs are provided to allow implementation of algorithms such as cascade control, temperature compensation of flows, and interlocking of logic inputs and continuous control loops.

A general purpose controller requires both continuous and sequential (logic) function blocks to be included in its library. In most industrial control applications, the performance of the LCU is adequate to sample all inputs, compute all of the function blocks, and generate all outputs in the range of 0.1 to 0.5 seconds maximum. Since configuration has such a small number of inputs, function blocks, and outputs, the performance requirement can be met easily by a simple and inexpensive set of microprocessor-based hardware (e g an eight-bit microprocessor and matching memory components). The communication requirements on configuration A for purposes of human interfacing are minimal since only one loop is controlled and a few input points monitored. However it is important that a secure inter controller communication channel be provided so that the single-loop controller can participate in complex control system structures that contains other LCUs.

An important architectural feature to be considered when evaluating industrial controllers is the provision for control output security in commercially available controllers, one or both of the following methods are used to permit continued operation of the process in the event of a controller failure.

(1) A backup feature is provided to allow the operator to adjust the control output manually if the automatic controller fails, or

(2) A redundant controller is provided to allow continuation of automatic control if the primary controller fails. The choice of method depends on the number of control outputs that would be lost if a controller fails.

An operator can handle a small number of loops (usually in the range of one to four) manually, that is so for small controllers usually only a manual backup is provided. In case of controller architecture A, the single-loop integrity of the controller configuration allows the simple and inexpensive option of manual backup to be used in most applications. For controllers that implement large numbers of control loops, some form of control redundancy is usually provided.

LCU Configuration B - In the first place, two different types of LCUs (continuous control and logic control are used to provide the full range of required controller functionality). In general increasing the number of types of controllers in the system has both positive and negative effects on the positive side, that is the specialized design of each controller allows it to match the functional needs of the corresponding application more closely than would a single general purpose controller. As a result, a particular control application would require a smaller number of controller hardware modules on the other hand, an increase in the number of controller types from one to two also results in increased interfacing requirements between controllers, additional documentation and training needs, and a decrease in production volume for each controller type (resulting in higher unit costs and longer lead times). In addition, this increase will reduce the scalability of the resulting system, since in the general case both of the controller types

will be needed even in the smallest system. The optimum tradeoff is dependent on the range of applications foreseen for the system being designed.
With respect to controller size, configuration B is medium in scale. The continuous control portion has the form of an eight-loop controller and the logic control portion can be viewed as a small programmable logic controller (PLC) or equivalent. Because of the split in functions and the relatively small size of each controller, the performance requirements on the controller hardware is to meet the 0.1 to 0.5 seconds cycle time are not excessive. This controller is usually implemented using a high performance eight bit or an average performance 16-bit microprocessor and matching memory components.

In the area of communication channels required, configuration B calls for a well designed interface between the two controller types, since in many systems both controllers must operate in close coordination to integrate continuous anti logic control functions. The ability to communicate with other controllers in the distributed system must be provided, but the communication performance level need not be as high as in configuration A since a greater percentage of communications takes place within the LCU. Of course, as with Configuration A, a communication channel to human interface devices has to be provided, but the channel must have a larger bandwidth because of the larger volume of traffic required per controller.
Finally, regarding output security, the lack of single-loop integrity of configuration B implies that a redundant controller must be used in all critical applications (i e , those in which a controller failure would cause a significant upset to the process). Of course if the application requires both continuous and logic control, redundancy must be provided for both of the controller types in the system. It has to be noted that full one-on-one redundancy often is not used in commercially available controllers. Instead, one redundant controller may be used to back up several primary controllers. This can reduce the cost of redundancy to some extent, but increases the complexity of the hardware used for interfacing the primary and backup controllers.

LCU Configuration C is closer to the structure to direct digital control (DDC) systems than the other two configurations. It is designed as multiloop controller in which all functions are performed by one CPU in conjunction with its associated memory and I/O boards. This places stringent requirements on the performance of the hardware since all of the control algorithms in the LCU must be executed within 0.5 seconds of less. This LCU configuration is implemented with one or more 16-bit microprocessors or a 32-bit microprocessor in conjunction with support hardware such as arithmetic coprocessors to attain the required speed.

In this configuration, it also becomes feasible that it include a high-level Language (usually BASIC or FORTRAN) in addition to or instead of function blocks. This architecture has a number of advantages over configurations A and B. For example, requirements for communication among controllers are minimal because of the high density of functions implemented per controller. Also this LCU's high-level language capability allows it to implement complex user-defined control and computational algorithms. However, it is not as scalable to small systems as the other two architectures, and the communication port or ports to the operator interface hardware must handle a large volume of traffic. In addition, since a failure of this type of LCU could affect a large number of control loops, are redundant CPU (and perhaps redundant I/O hardware as

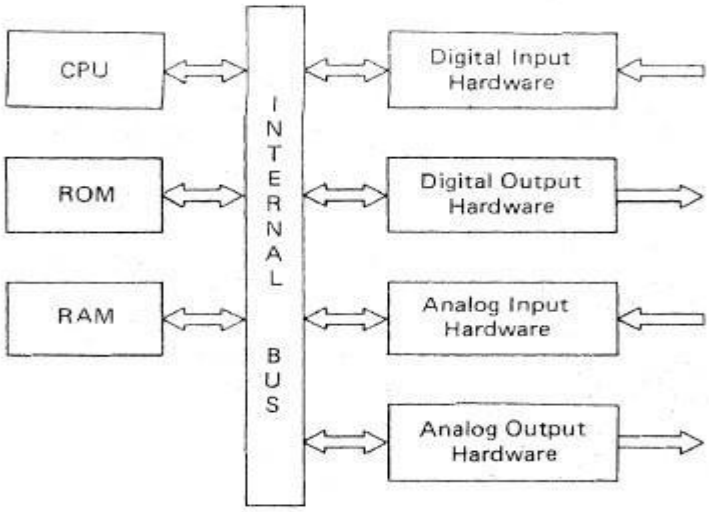
well). Redundant hardware may not be required if only a few loops are affected by a failure or if the controller is performing only high-level control functions and is not manipulating control outputs directly.

3.5 LOCAL CONTROL UNIT (LCU)

* Smallest collection of hardware that performs closed loop control and interfaces directly with the process.
* It takes input from process measuring devices and commands from operator.
* Computes the control outputs needed to make process follow the command.
* Sends control output to actuators, drives valves and other mechanical devices.

Requirements of LCU

* Flexibility of changing the control configuration.
* Ability to use the controller without being a computer expert.
* Ability of the LCU to communicate with other elements in the system.
* Ability to bypass the failed controller manually.



Basic Elements of a Local Control Unit.

Fig.3.7 Basic Elements of a Local Control Unit

The LCU also have input/output circuitry so that it can communicate with the external world by reading in or receiving, analog and digital data as well as sending similar signals out. Generally, the CPU communicates with the other elements in the LC U over an internal shared bus that transmits addressing, data control, and status information in addition to the data .The controller structure shown in figure is the minimum that required to perform basic control functions. In a noncritical application in which the control function never changes, this structure might be adequate. The control algorithms could be coded in assembly language and loaded into ROM. After the controller was turned on, it would read inputs, execute the control algorithms, and generate control outputs in a fixed cycle in definitely. However, because the situation is

not this simple in industrial control applications, the controller structure shown in Figure must be enhanced to include the following:

1. Flexibility of changing the control configuration -In industrial applications the same controller product usually is used to implement a great variety of different control strategies. Even for a particular strategy, the user usually wants the flexibility of changing the control system tuning parameters without changing the controller hardware. Therefore the control configuration cannot be burned into ROM but must be stored in a memory medium whose contents can be changed, such as RAM. Unfortunately, RAM is usually implemented using semiconductor technology that is volatile that is, it loses its contents if the power is turned off (whether due to power failure, routine maintenance or removal of the controller from its cabinet). Therefore, some provision must be made for restoring the control configuration, either from an external source or from a nonvolatile memory within the controller itself.
2. Ability to use the controller without being a computer expert-The typical user of an industrial control system is generally familiar with the process to be controlled, knows the basics of control system design, and has worked with electric analog or pneumatic control systems before. However, the user is usually not capable of or interested in programming a microprocessor in assembly language. He or she simply wants to be able to implement the selected control algorithms. Therefore a mechanism for allowing the user to ‘ configure” the LCU’s control algorithms in a relatively simple way must be provided.
3. Ability to bypass the controller in case it fails so that the process still can be controlled manually- Shutting down the process is very expensive and undesirable for the control system user. Since all control equipment has the potential of failing no matter how carefully it has been designed, the system architecture must allow an operator to “take over” the control loop and run it by hand until the control hardware is repaired or replaced.
4. Ability of the LCU to communicate with other LCUs and other elements in the system-Controllers in an industrial control system do not operate in isolation but must work in conjunction with other controllers, devices, and human interface devices.

3.6 Redundant Controller Designs

The manual backup approach to control system security is viable if the LCU in question handles only a few loops and if the process being controlled is relatively slow. On the other hand situations, however, require some form of controller redundancy to ensure that automatic control of the process is maintained in spite of an LCU failure.

By their nature, redundant control system structures are very complex than those that rely on manual backup. The addition of redundant elements to the basic control system will always result in an increase in system cost and also in additional maintenance to service the extra hardware. The redundant structure must be designed carefully to ensure that system reliability actually increases enough to offset these drawbacks. Some of the guidelines to follow in evaluating or designing a redundant control system are given as follows.

The redundant architecture should be kept as simple as possible as there is a law of diminishing returns in redundancy design. At some point, adding more hardware will reduce system reliability.

1. As much as possible, the architecture must minimize single points of failure. The redundant hardware elements must be in dependent as possible so that the failure of any one does not

bring the rest down as well (in each design, however, there is always at least one single point of failure; this element must be designed to be as simple and reliable as possible.)

2. Also, the redundant nature of the controller configuration should be transparent to the user; that is, the user should be able to deal with the redundant system in the same way as a non redundant one. This includes both operational and engineering functions (e.g., control system configuration and tuning). If one of the redundant elements fails, the steps to follow in repairing and restarting the system is to be clear to the user.
3. The process should not be bumped or disturbed either when one of the redundant elements fails or when the user puts the repaired element back on line.
4. After a control element has failed, the system should not have to rely on it to perform any positive action or to provide any necessary information to other elements in the system until after repair or replacement .
5. The redundant LCU architecture must have the capability for "hot" spare replacement; that is, allow for the replacement of failed redundant elements without shutting down the total LCU. The following discussion will describe, in order to increase complexity, several approaches for designing a redundant LCU architecture:
   - CPU redundancy
   - One-on-one redundancy
   - One-on-many redundancy
   - Multiple active redundancy

In each case, the key advantages and disadvantages of the approach will be listed. No attempt will be made to recommend a best approach; as always, this will depend on the particular control system application.

3.6.1 CPU Redundancy

Only one redundant element will be active at a time. The backup element takes over if the primary fails. In the first configuration, only the CPU portion of the LCU is redundant, but the I/O circuitry will not be redundant. This configuration is more popular in areas where large number of control loops have to be implemented. Only one of the CPUs is active which performs operations such as reading inputs, performing control computations, and generating control outputs at any one time. The user designates the primary CPU, through the priority arbitrator circuit shown in Figure, using a switch setting mechanism. After startup, the arbitrator starts monitoring the operation of the primary CPU and if the arbitrator detects a failure in the primary , it will transfer the priority to the backup. During this operation, the backup CPU will periodically update its internal memory by reading the state of the primary CPU through the arbitrator .Though, both CPUs are connected to the plant communication system, only the primary will be active in transmitting and receiving messages over this link. The operator and engineering interface used in this system is the high-level human interface which is usually a CRT-

based video display unit that interfaces with the LCU over the shared communication facilities. Only the primary CPU will accept control commands and tuning changes transmitted by the VDU. The primary CPU, in turn, updates the backup CPU with this updated information. By following the same methodology, all monitoring and status information in the LCU is transmitted to the VDU by the primary CPU.

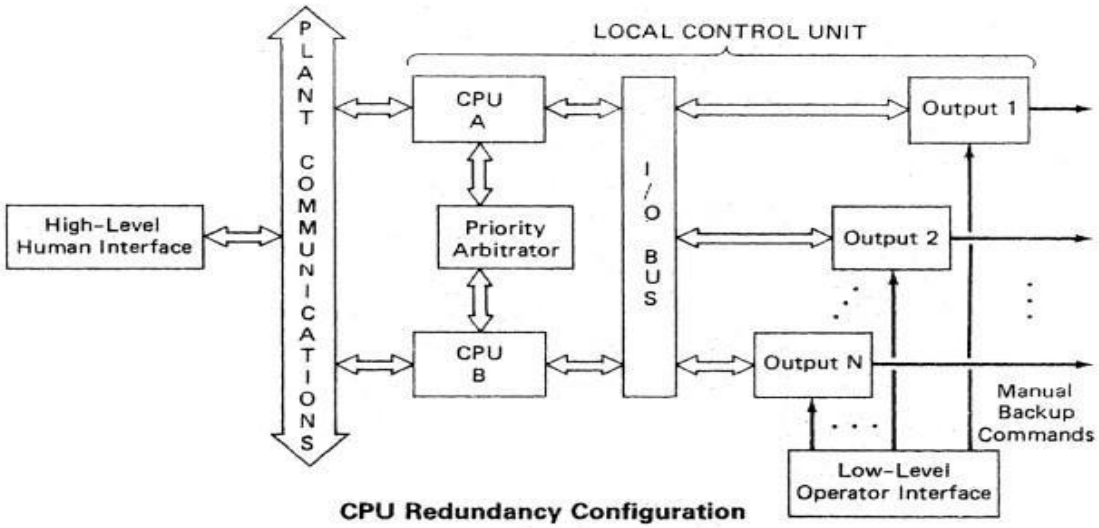


Fig.3.8 CPU Redundancy Configuration

The key advantages of this approach- relatively easy to implement and cost-effective. The redundant elements can interface easily to the plant communication facilities and the 110 bus, which are shared communication channels. The cost of the redundant hardware will not be excessive since only the CPU hardware is duplicated. Problems may occur with this architecture if it is not designed properly. For example, the I/O bus and the priority arbitrator will represent potential single points of failure in the configuration. They must be designed in such a way that their failure does not affect both CPUs . Another potential problem is that the low-level operator interface is physically located near the LCU.

3.6.2 One-on-One Redundancy

The remaining three redundancy approaches provides redundancy in the control output circuit as well as in the CPU hardware. Hence, most of these architectures do not support a low-level operator interface for manual backup purposes. This approach shown in Figure provides a total backup LCU to the primary LCU . An output switching block must be included to transfer the outputs when the controller fails, since the control output circuit is duplicated . Similar to first redundant configuration a priority arbitrator designates the primary and backup LCUs and activates the backup if a failure in the primary is detected. It also serves as the means to update the internal states of the backup

LCU . In this configuration, the arbitrator has the additional responsibility of sending a command to the output switching circuitry if the primary LCU fails, causing the backup LCU to generate the control outputs Communications with the high-level human interface which are handled in the same way as in the CPU-redundant configuration.

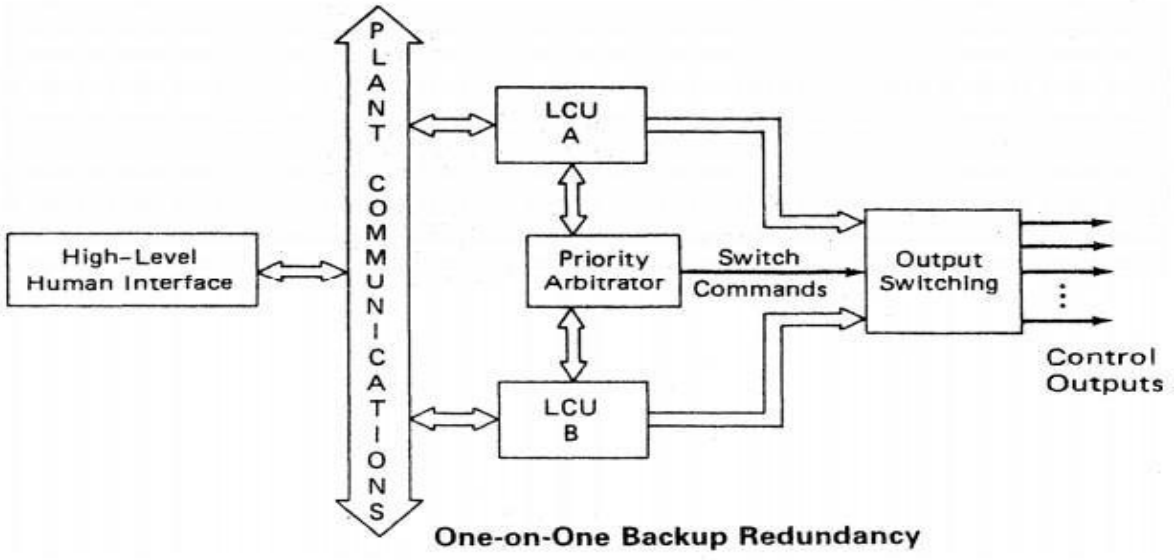


Fig.3.9 One –on – One Backup Redundancy

The main advantage of the one-on-one configuration are no manual backup is needed. It eliminates any questions that may arise with a partial redundancy approach , particularly regarding the decision on which elements are to be redundant. However, it also suffers from a number of disadvantages . First, it is an expensive approach to redundancy, but additional equipments must be included to manage the redundant ones. It also suffers from potential single-point failure problems with the arbitrator and the output switching circuitry.

3.6.3 One-on -Many Redundancy

In this configuration, a single LCU is used as a standby to back up any one of several primary LCUs. An arbitrator is required to monitor the status of the primary controller and switch to the backup when a failure occurs. Unfortunately, there is no way to know beforehand, for which primary controller the backup has to replace. Hence, a very general switching matrix is necessary to transfer the I/O from the failed controller to the backup. It is also not possible to preload the backup controller with the control system configuration for any particular primary LCU. Rather, the configuration is loaded into the backup LCU from the primary LCU only after the primary has failed. The cost of this approach is lower than the other three redundancy configurations because only a small portion of the control hardware is duplicated. The switching matrix is an element whose operation is essential to control the loops concerned, but it also represents a potential single point failure. Because of this complexity, it must be designed very carefully so that its failure does not affect the other loops. Second, the approach relies on the failed controller to provide a copy of the control system configuration to the backup LCU. A better approach would be provided if we store a copy of each primary LCU’s control

configuration in the arbitrator. When an LCU failure occurs, the arbitrator could then load the proper configuration into the backup LCU.

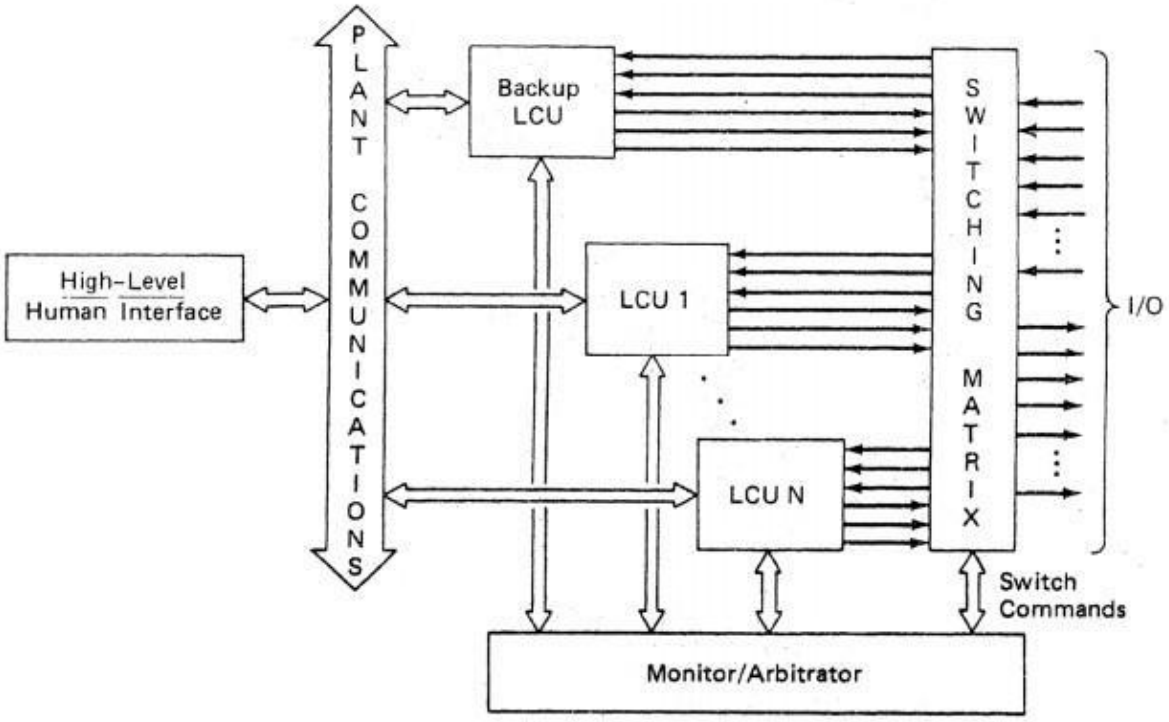


Fig.3.10 Multiple Active Redundancy

In this approach, three or more redundant LCUs are used to perform the same control functions. In this one of the redundant controllers will be active at the same time in reading process inputs, computing the control calculations, and generating control outputs to the process. Each LCU has access to all of the process inputs needed to implement the control configuration. An output voting device selects one of the valid control outputs from the controllers and transmits it to the control process. When a controller fails, it is designed to generate an output outside the normal range. The output voting device will then discard this output as an invalid one. In the case of analog control outputs, the output voting device, is often designed to select the median signal; in the case of digital control outputs, the voting device is designed to select the signal generated by at least two out of the three controllers. Each controller has access to the output of the voting device to check its own operation and shut down if its output disagrees significantly with that of the other controllers.

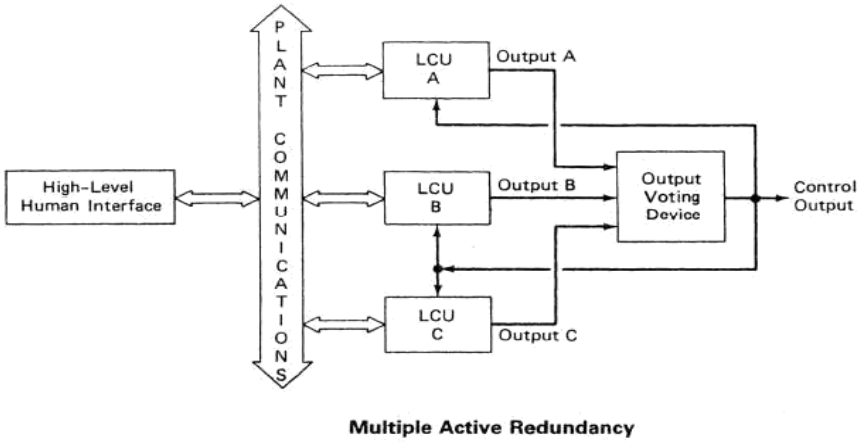


Fig.3.11 Multiple Active Redundancy

The multiple active redundancy configuration is a very sophisticated approach which is very complex and obviously costs three times more than a non redundant system. In its modified form it has been used in safety systems for nuclear power plants. More often, this configuration has been used in high-reliability computer control applications such as aerospace industry . This approach may find its way into selected process control applications as hardware costs continue to decline and configurations become standardized. The main advantage of this approach is that, as long as the output voting device is designed for high reliability, it significantly increases the reliability of the control system. However, this architecture suffers various disadvantages in addition to cost. In most of the applications where this configuration has been used so far, the approach has implemented a fixed configuration. The process of ensuring that control system configuration and tuning changes have been implemented properly is not trivial. Other drawbacks of this approach are that the added hardware requires increased maintenance and also the system is very complex.

UNIT- III Part- A

Sl.no	Part A - Questions	CO	Level
1	Define DCS	CO2	L1
2	Write few applications of DCS	CO2	L6
3	Point out the various elements in DCS	CO2	L4
4	Discuss the significance of LLHI	CO2	L2
5	List the various architecture of DCS	CO2	L1
6	Classify the various redundant controllers in DCS	CO2	L1
7	Define redundant controller	CO2	L1
8	Tell the functions of LCU	CO2	L1
9	Compare LLHI and HLHI	CO2	L5
10	Report the architecture parameter to be considered while selecting LCU	CO2	L3

UNIT- III Part – B

Sl.no	Part B - Questions	CO	Level
1	Explain the generalized architecture of DCS with neat sketch	CO2	L2
2	Discuss and compare the different architecture of DCS	CO2	L2
3	Sketch and discuss about local control unit in DCS	CO2	L3
4	Explain and compare the various redundancy controllers	CO2	L2
5	Compare Hybrid and central computer system architecture with construction and working	CO2	L2
6	List different configurations of LCU, explain the various blocks in LCU	CO2	L1, L2

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT-V-VIRTUAL BIOINSTRUMENTATION – **SEIA1503**

I Introduction

Virtual instrumentation - Definition, flexibility - Block diagram and architecture of virtual instruments - Virtual instrumentsversus traditional instruments -LabVIEW software and interfacing with Hardware, VI programming techniques - VI, sub VI,loops and charts, arrays, clusters and graphs, interfacing LabView with Matlab .-LabVIEW applications in Instrumentation.

INTRODUCTION

An instrument is a device designed to collect data from an environment, or from a unit under test, and to display information to a user based on the collected data. Such an instrument may employ a transducer to sense changes in a physical parameter, such as temperature or pressure, and to convert the sensed information into electrical signals, such as voltage or frequency variations. The term instrument may also be defined as a physical software device that performs an analysis on data acquired from another instrument and then outputs the processed data to display or recording devices. This second category of recording instruments may include oscilloscopes, spectrum analyzers, and digital millimeters. The types of source data collected and analyzed by instruments may thus vary widely, including both physical parameters such as temperature, pressure, distance, frequency and amplitudes of light and sound, and also electrical parameters including voltage, current, and frequency.

Virtual instrumentation is an interdisciplinary field that merges sensing, hardware, and software technologies in order to create flexible and sophisticated instruments for control and monitoring applications. The concept of virtual instrumentation was born in late 1970s, when microprocessor technology enabled a machine's function to be more easily changed by changing its software. The flexibility is possible as the capabilities of a virtual instrument depend very little on dedicated hardware – commonly, only application specific signal conditioning module and the analog-to- digital converter used as interface to the external world. Therefore, simple use of computers or specialized onboard processors in instrument control and data acquisition cannot be defined as virtual instrumentation. Increasing number of biomedical applications use virtual instrumentation to improve insights into the underlying nature of complex phenomena and reduce costs of medical equipment and procedures.

History of Instrumentation Systems

Historically, instrumentation systems originated in the distant past, with measuring rods, thermometers, and scales. In modern times, instrumentation systems have generally consisted of individual instruments, for example, an electromechanical pressure gauge comprising a sensing transducer wired to signal conditioning circuitry, outputs a processed signal to a display panel and

perhaps also to a line recorder, in which a trace of changing conditions is linked onto a rotating drum by a mechanical arm, creating a time record of pressure changes. Complex systems such as chemical process control applications employed until the 1980s consisted of sets of individual physical instruments wired to a central control panel that comprised an array of physical data display devices such as dials and counters, together with sets of switches, knobs, and buttons for controlling the instruments. A history of virtual instrumentation is characterized by continuous increase of flexibility and scalability of measurement equipment. Starting from first manual-controlled vendor- defined electrical instruments, the instrumentation field has made a great progress toward contemporary computer-controlled, user-defined, sophisticated measuring equipment. Instrumentation had the following phases:

– Analog measurement devices

– Data acquisition and processing devices

– Digital processing based on general purpose computing platform

– Distributed virtual instrumentation

The first phase is represented by early “pure” analog measurement devices, such as oscilloscopes or EEG recording systems. They were completely closed dedicated systems, which included power suppliers, sensors, translators, and displays. They required manual settings, presenting results on various counters, gauges, CRT displays, or on the paper. Further use of data was not part of the instrument package, and an operator had to physically copy data to a paper notebook or a data sheet. Performing complex or automated test procedures was rather complicated or impossible, as everything had to be set manually.

Second phase started in 1950s, as a result of demands from the industrial control field. Instruments incorporated rudiment control systems, with relays, rate detectors, and integrators. That led to creation of proportional– integral–derivative (PID) control systems, which allowed greater flexibility of test procedures and automation of some phases of measuring process. Instruments started to digitalize measured signals, allowing digital processing of data, and introducing more complex control or analytical decisions. However, real-time digital processing requirements were too high for any but an onboard special purpose computer or digital signal processor (DSP). The instruments still were standalone vendor defined boxes.

In the third phase, measuring instruments became computer based. They began to include interfaces that enabled communication between the instrument and the computer. This relationship started with the general-purpose interface bus (GPIB) originated in 1960s by Hewlett-Packard (HP), then called HPIB, for purpose of instrument control by HP computers. Initially, computers were primarily used as off-line instruments. They were further processing the data after first recording the measurements on disk or type. As the speed and capabilities of general-purpose computers advanced exponentially general-purpose computers became fast enough for complex real-time measurements. It soon became possible to adapt standard, by now high-speed computers, to the online applications required in real-time measurement and control. New general-purpose computers

from most manufactures incorporated all the hardware and much of the general software required by the instruments for their specific purposes. The main advantages of standard personal computers are low price driven by the large market, availability, and standardization. Although computers' performance soon became high enough, computers were still not easy to use for experimentalists.

Nearly all of the early instrument control programs were written in BASIC, because it had been the dominant language used with dedicated instrument controllers. It required engineers and other users to become programmers before becoming instrument users, so it was hard for them to exploit potential that computerized instrumentation could bring. Therefore, an important milestone in the history of virtual instrumentation happened in 1986, when National Instruments introduced LabVIEW 1.0 on a PC platform. LabVIEW introduced graphical user interfaces and visual programming into computerized instrumentation, joining simplicity of a user interface operation with increased capabilities of computers. Today, the PC is the platform on which most measurements are made, and the graphical user interface has made measurements user-friendlier. As a result, virtual instrumentation made possible decrease in price of an instrument. As the virtual instrument depends very little on dedicated hardware, a customer could now use his own computer, while an instrument manufactures could supply only what the user could not get in the general market.

The fourth phase became feasible with the development of local and global networks of general purpose computers. Since most instruments were already computerized, advances in telecommunications and network technologies made possible physical distribution of virtual instrument components into telemedical systems to provide medical information and services at a distance.

Possible infrastructure for distributed virtual instrumentation includes the Internet, private networks, and cellular networks, where the interface between the components can be balanced for price and performance.

The introduction of computers into the field of instrumentation began as a way to couple an individual instrument, such as a pressure sensor, to a computer, and enable the display of measurement data on virtual instrument panel on the computer screen using appropriate software. The instrumental so contained buttons for controlling the operation of the sensor. Thus, such instrumentation software enabled the creation of a simulated physical instrument, having the capability to control physical sensing components.

Virtual Instrumentation

Virtual instrumentation achieved mainstream adoption by providing a new model for building measurement and automation systems. Keys to its success include rapid PC advancement; explosive low-cost, high-performance data converter (semiconductor) development; and system design software emergence. These factors make virtual instrumentation systems accessible to a very broad base of users.

Definition

A virtual instrumentation system is software that is used by the user to develop a computerized test and measurement system, for controlling an external measurement hardware device from a desktop computer, and for displaying test or measurement data on panels in the computer screen. The test and measurement data are collected by the external device interfaced with the desktop computer. Virtual instrumentation also extends to computerized systems for controlling processes based on the data collected and processed by a PC based instrumentation system.

Block diagram and architecture of a virtual instrument

A virtual instrument is composed of the following blocks:

- Sensor module
- Sensor interface
- Information systems interface
- Processing module
- Database interface
- User interface

Figure shows the general architecture of a virtual instrument. The sensor module detects physical signal and transforms it into electrical form, conditions the signal, and transforms it into a digital form for further manipulation. Through a sensor interface, the sensor module communicates with a computer. Once the data are in a digital form on a computer, they can be processed, mixed, compared, and otherwise manipulated, or stored in a database.

Then, the data may be displayed, or converted back to analog form for further process control. Virtual instruments are often integrated with some other information systems. In this way, the configuration settings and the data measured may be stored and associated with the available records. In following sections each of the virtual instruments modules are described in more detail.

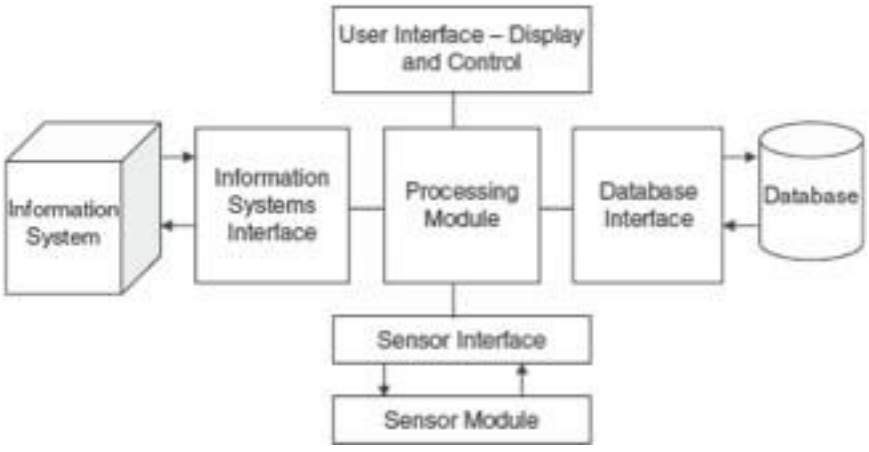


Fig 1: Architecture of a virtual instrument

- **Sensor Module**

The sensor module performs signal conditioning and transforms it into a digital form for further manipulation. Once the data are in a digital form on a computer, they can be displayed, processed, mixed, compared, stored in a database, or converted back to analog form for further process control. The database can also store configuration settings and signal records. The sensor module interfaces a virtual instrument to the external, mostly analog world transforming measured signals into computer readable form. A sensor module principally consists of three main parts:

- The sensor
- The signal conditioning part
- The A/D converter

The sensor detects physical signals from the environment. If the parameter being measured is not electrical, the sensor must include a transducer to convert the information to an electrical signal, for example, when measuring blood pressure.

The signal-conditioning module performs (usually analog) signal conditioning prior to AD conversion. This module usually does the amplification, transducer excitation, linearization, isolation, or filtering of detected signals.

The A/D converter changes the detected and conditioned voltage into a digital value. The converter is defined by its resolution and sampling frequency. The converted data must be precisely time- stamped to allow later sophisticated analyses. Although most biomedical sensors are specialized in processing of certain signals, it is possible to use generic measurement components, such as data acquisition (DAQ), or image acquisition (IMAQ) boards, which may be applied to broader class of signals. Creating generic measuring board, and incorporating the most important components of different sensors into one unit, it is possible to perform the functions of many medical instruments on the same computer.

- **Sensor Interface**

There are many interfaces used for communication between sensors modules and the computer. According to the type of connection, sensor interfaces can be classified as wired and wireless.Wired Interfaces are usually standard parallel interfaces, such as GPIB, Small Computer Systems Interface (SCSI), system buses (PCI eXtension for Instrumentation PXI or VME Extensions for Instrumentation (VXI), or serial buses (RS232 or USB interfaces).Wireless Interfaces are increasingly used because of convenience. Typical interfaces include 802.11 family of standards, Bluetooth, or GPRS/GSM interface. Wireless communication is especially important for implanted sensors where cable connection is impractical or not possible. In addition, standards, such as Bluetooth, define a self-identification protocol, allowing the network to configure dynamically and describe itself. In this way, it is possible to reduce installation cost and create plug-and-play like networks of sensors. Device miniaturization allowed development of Personal Area Networks (PANs) of intelligent sensors Communication with medical devices is also standardized with the IEEE 1073 family of standards. This interface is intended to be highly robust in an environment

where devices are frequently connected to and disconnected from the network.

- **Processing Module**

Integration of the general purpose microprocessors/microcontrollers allowed flexible implementation of sophisticated processing functions. As the functionality of a virtual instrument depends very little on dedicated hardware, which principally does not perform any complex processing, functionality and appearance of the virtual instrument may be completely changed utilizing different processing functions. Broadly speaking, processing function used in virtual instrumentation may be classified as analytic processing and artificial intelligence techniques.

Analytic functions define clear functional relations among input parameters. Some of the common analyses used in virtual instrumentation include spectral analysis, filtering, windowing, transforms, peak detection, or curve fitting. Virtual instruments often use various statistics function, such as, random assignment and bio statistical analyses. Most of those functions can nowadays be performed in real-time.

. Artificial intelligence technologies could be used to enhance and improve the efficiency, the capability, and the features of instrumentation in application areas related to measurement, system identification, and control. These techniques exploit the advanced computational capabilities of modern computing systems to manipulate the sampled input signals and extract the desired measurements. Artificial intelligence technologies, such as neural networks, fuzzy logic and expert systems, are applied in various applications, including sensor fusion to high- level sensors, system identification, prediction, system control, complex measurement procedures, calibration, and instrument fault detection and isolation. Various nonlinear signal processing, including fuzzy logic and neural networks, are also common tools in analysis of biomedical signals. Using artificial intelligence it is even possible to add medical intelligence to ordinary user interface devices. For example, several artificial intelligence techniques, such as pattern recognition and machine learning, were used in a software-based visual-field testing system.

- **Database Interface**

Computerized instrumentation allows measured data to be stored for off-line processing, or to keep records as a part of the patient record. There are several currently available database technologies that can be used for this purpose. Simple usage of file systems interface leads to creation of many proprietary formats, so the interoperability may be a problem. The eXtensible Markup Language (XML) may be used to solve interoperability problem by providing universal syntax. The XML is a standard for describing document structure and content. It organizes data using markup tags, creating self-describing documents, as tags describe the information it contains. Contemporary database management systems such SQL Server and Oracle support XML import and export of data. Many virtual instruments use DataBase Management Systems(DBMSs). They provide efficient management of data and standardized insertion, update, deletion, and selection. Most of

these DBMSs provided Structured Query Language (SQL) interface, enabling transparent execution of the same programs over database from different vendors. Virtual instruments usethese DMBSs using some of programming interfaces, such as ODBC, JDBC,ADO, and DAO.

- **Information System Interface**

Virtual instruments are increasingly integrated with other medical information systems, such as hospital information systems. They can be used to create executive dashboards, supporting decision support, real time alerts, and predictive warnings. Some virtual interfaces toolkits, such as LabVIEW, provide mechanisms for customized components, such as ActiveX objects, that allows communication with other information system, hiding details of the communication from virtual interface code. In Web based applications this integration is usually implemented using Unified Resource Locators (URLs). Each virtual instrument is identified with its URL, receiving configuration settings via parameters. The virtual instrument then can store the results of the processing into a database identified with its URL.

- **Presentation and Control**

An effective user interface for presentation and control of a virtual instrument affects efficiency and precision of an operator do the measurements and facilitates result interpretation. Since computer's user interfaces are much easier shaped and changed than conventional instrument's user interfaces, it is possible to employ more presentation effects and to customize the interface for each user. According to presentation and interaction capabilities, we can classify interfaces used in virtual instrumentation in four groups:

- Terminal user interfaces
- Graphical user interfaces
- Multimodal user interfaces and
- Virtual and augmented reality interfaces

Terminal User Interfaces

First programs for instrumentation control and data acquisition had character-oriented terminal user interfaces. This was necessary as earlier general-purpose computers were not capable of presenting complex graphics. As terminal user interfaces require little of system resources, they were implemented on many platforms. In this interfaces, communication between a user and a computer is purely textual. The user sends requests to the computer typing commands, and receives response in a form of textual messages.

Graphical User Interfaces

Graphical user interfaces (GUIs) enabled more intuitive human–computerinteraction, making virtual instrumentation more accessible. Simplicity ofinteraction and high intuitiveness of graphical user interface operations madepossible creation of user-friendlier virtual instruments. GUIs allowed creationof many sophisticated graphical widgets such as graphs, charts, tables, gauges,or meters, which can easily be created with many user interface tools.

Multimodal Presentation

In addition to graphical user interfaces that improve visualization, contemporarypersonal computers are capable of presenting other modalities suchas sonification or haptic rendering.

Multimodal combinations of complementarymodalities can greatly improve the perceptual quality of user interfaces.

Virtual Instruments versus Traditional Instruments

Stand-alone traditional instruments such as oscilloscopes and waveform generators are very powerful, expensive, and designed to perform one or more specific tasks defined by the vendor. However, the user generally cannot extend or customize them. The knobs and buttons on the instrument, the built-in circuitry, and the functions available to the user, are all specific to the nature of the instrument. In addition, special technology and costly components must be developed to build these instruments, making them very expensive and slowto adapt.

Table 1: Traditional Instruments Vs Virtual Instruments

Traditional Instruments	Virtual Instruments
Vendor defined	User-defined
Function specific,stand alone with limited connectivity	Application oriented system with connectivity to networks,peripherals, and applications
Hardware is the key	Software is the key
Expensive	Low cost , reusable
Closed, fixed functionality	Open, flexible functionality
Slow turn on technology	Fast turn on technology
Minimal economics of scale	Maximum economics of scale
High development and maintenance cost	Software minimizes development and maintenance cost

Advantages of VI

The virtual instruments running on notebook automatically incorporate their portable nature to the Engineers and scientists whose needs, applications and requirements change very quickly, need flexibility to create their own solutions. We can adapt a virtual instrument to our particular needs without having to replace the entire device because of the application software installed on the PC and the wide range of available plug-in hardware.

Performance

In terms of performance, LabVIEW includes a compiler that produces native code for the CPU platform. The graphical code is translated into executable machine code by interpreting the syntax and by compilation. The LabVIEW syntax is strictly enforced during the editing process and compiled into the executable machine code when requested to run or upon saving. In the latter case, the executable and the source code are merged into a single file. The executable runs with the help of the LabVIEW run-time engine, which contains some precompiled code to perform common tasks that are defined by the G language. The run-time engine reduces compile time and also provides a consistent interface to various operating systems, graphic systems, hardware components, etc.

Platform-Independent Nature

A benefit of the LabVIEW environment is the platform-independent nature of the G-code,

which is (with the exception of a few platform specific functions) portable between the different LabVIEW systems for different operating systems (Windows, MacOSX, and Linux). National Instruments is increasingly focusing on the capability of deploying LabVIEW code onto an increasing number of targets including devices like Pharlap OS-based LabVIEW realtime controllers, PocketPCs, PDAs, Fieldpoint modules, and into FPGAs on special boards.

Flexibility

Except for the specialized components and circuitry found in traditional instruments, the general architecture of stand-alone instruments is very similar to that of a PC-based virtual instrument. Both require one or more microprocessors, communication ports (for example, serial and GPIB), and display capabilities, as well as data acquisition modules. These devices differ from one another in their flexibility and the fact that these devices can be modified and adapted to the particular needs.

Lower Cost

By employing virtual instrumentation solutions, lower capital costs, system development costs, and system maintenance costs are reduced, increasing the time to market and improving the quality of our own products.

Plug-In and Networked Hardware

There is a wide variety of available hardware that can either be plugged into the computer or accessed through a network. These devices offer a wide range of data acquisition capabilities at a significantly lower cost than that of dedicated devices.

The Costs of a Measurement Application

The costs involved in the development of a measurement application can be divided into five distinct areas composed of hardware and software prices and several time costs. The price of the hardware and software was considered as the single largest cost of their most recent test or measurement system. However, the cumulative time costs in the other areas make up the largest portion of the total application cost.

Reducing System Specification Time Cost

Deciding the types of measurements to take and the types of analysis to perform takes time. Once the user has set the measurement specifications, the user must then determine exactly the method to implement the measurement system. The time taken to perform these two steps equals the system specification time.

Lowering the Cost of Hardware and Software

The price of measurement hardware and software is undoubtedly the most visible cost of a data-acquisition system. Many people attempt to save money in this area without considering the effect on the total development cost.

Minimizing Set-Up and Configuration Time Costs

Once the users have specified and purchased measurement hardware, the real task of developing the application begins. However, the user must first install the hardware and software, configure any necessary setting and ensure that all pieces of the system function properly.

Dataflow Programming

Lab VIEW follows a dataflow model for running VIs. A block diagram node executes when all its inputs are available. When a node completes execution, it supplies data to its output terminals and passes the output data to the next node in the dataflow path. Visual Basic, C + +, JAVA, and most other text-based programming languages follow a control flow model of program execution. In control flow, the sequential order of program elements determines the execution order of a program. For a dataflow-programming example, consider a block diagram that adds two numbers and then subtracts 50.00 from the result of the addition as illustrated in Fig. In this case, the block diagram executes from left to right, not because the objects are placed in that order, but because the Subtract function cannot execute until the Add function finishes executing and passes the data to the Subtract function. Remember that a node executes only when data are available at all of its input terminals, and it supplies data to its output terminals only when it finishes execution.

In the following example in Fig. 2.consider which code segment would execute first – the Add, Random Number, or Divide function. In a situation where one code segment must execute before another and no data dependency exists between the functions, the user can use other programming methods, such as error clusters, to force the order of execution.

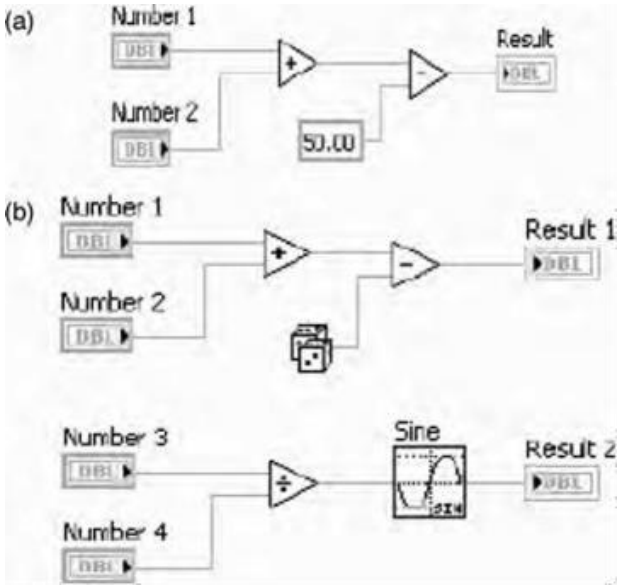


Fig. 2. Dataflow programming example

'G' Programming

The 'G' sequence structure is used to control execution order when natural data dependency does not exist. The user also can create an artificial data dependency in which the receiving node does not actually use the data received. Instead, the receiving node uses the arrival of data to trigger its execution. The programming language used in LabVIEW, called "G", is a dataflow language. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, "G" is inherently capable of parallel execution. Multiprocessing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution. Programmers with a

background in conventional programming often show a certain reluctance to adopt the LabVIEW dataflow scheme, claiming that LabVIEW is prone to race conditions. In reality, this stems from a misunderstanding of the data-flow paradigm. The afore-mentioned data-flow (which can be "forced," typically by linking inputs and outputs of nodes) completely defines the execution sequence, and that can be fully controlled by the programmer. The graphical approach also allows nonprogrammers to build programs by simply dragging and dropping virtual representations of the lab equipment with which they are already familiar. The LabVIEW programming environment, with the included examples and the documentation, makes it easy to create small applications. This is a benefit on one side but there is also a certain danger of underestimating the expertise needed for good quality "G" programming. For complex algorithms or large-scale code it is important that the programmer understands the special LabVIEW syntax and the topology of its memory management well. The most advanced Lab-VIEW development systems offer the possibility of building standalone applications.

Virtual Instruments

Virtual Instruments are front panel and block diagram. The front panel or user interface is built with controls and indicators. Controls are knobs, pushbuttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays.

Front Panel

The front panel is the window through which the user interacts with the program. The input data to the executing program is fed through the front panel and the output can also be viewed on the front panel, thus making it indispensable.

Front Panel Toolbar controls and Indicators

The front panel is primarily a combination of controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates.

S. No	Icon	Name	Meaning
1		**Run**	Used to run a VI. The VI runs if the **Run** button appears as a solid white arrow as shown at the left. The solid white arrow also indicates the VI can be used as a subVI if a connector pane for the VI is created.
2		**Run**	While the VI runs, the **Run** button appears as shown at left if the VI is a top-level VI, meaning it has no callers and therefore is not a subVI.
3		**Run**	If the VI that is running is a subVI, the **Run** button appears as shown at left.
4		**Run**	The **Run** button appears broken, shown at left, when the VI that is created and edited contains errors. If the **Run** button still appears broken after wiring of the block diagram is completed, then the VI is broken and cannot run. On clicking this button, an **Error list** window is displayed, which lists all errors and warnings.
5		**Run Continuously**	The **Run Continuously** button, shown at left, is used to run the VI until one abortion or pause execution.
6		**Abort Execution**	While the VI runs, the **Abort Execution** button, shown at left, appears. Click this button to stop the VI immediately if there is no other way to stop the VI. If more than one running top-level VI uses the VI, the button is dimmed.
7		**Pause**	Click the **Pause** button, shown at left, to pause a running VI. Upon clicking the **Pause** button, LabVIEW highlights on the block diagram the location where the execution was paused, and the **Pause** button appears red. Click the button again to continue running the VI.

Fig 3: Front panel tool bars

Controls and Indicators :

Controls and Indicators are generally not interchangeable; the difference should be clear among the user. The user can "drop" controls and indicators onto the front panel by selecting them from a subpalette of the floating **Controls** palette window shown in Figure.

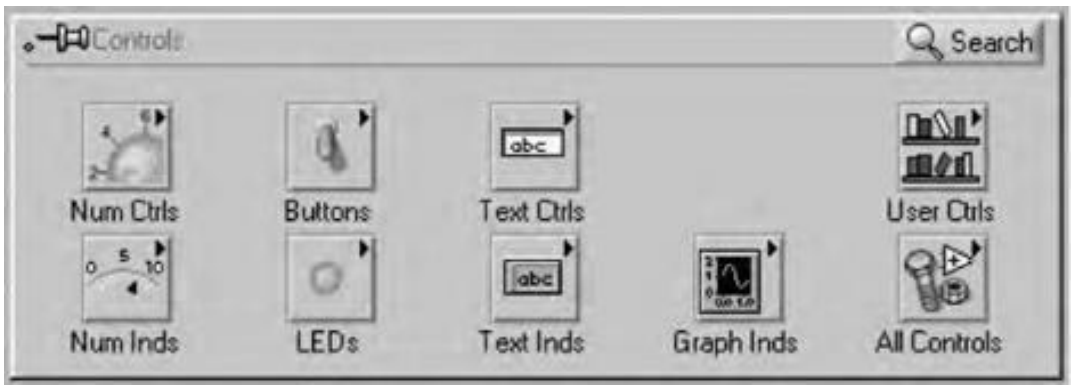


Fig 4: Controls and Indicators

Once an object is on the front panel, the user can easily adjust its size, shape, position, color, and other attributes. Controls and indicators can be broadly classified as:

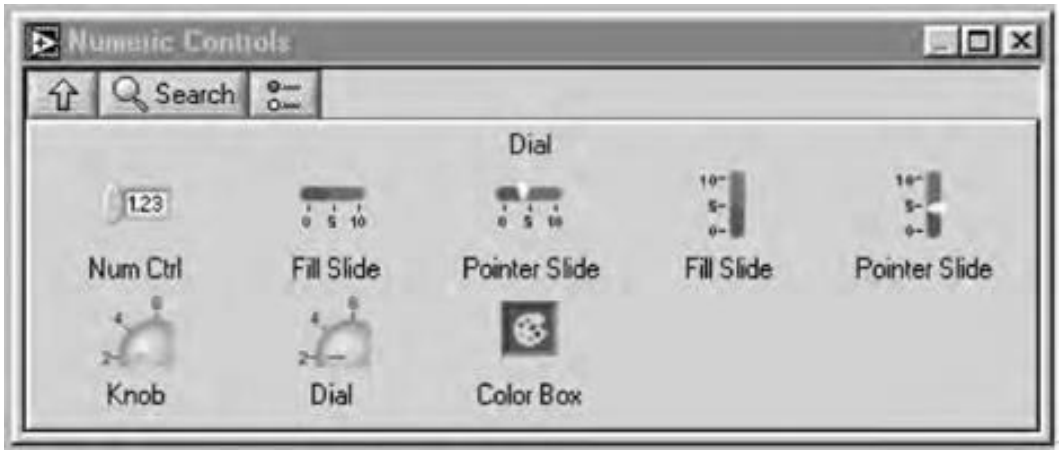
- Numeric controls and indicators
- Boolean controls and indicators

Numeric Controls and Indicators

The two most commonly used numeric objects are the numeric control and the numeric indicator, as shown in Fig. The values in a numeric control can be entered or changed by selecting the increment and decrement buttons with the Operating tool or double-clicking the number with either the Labeling tool or the Operating tool.

Boolean Controls and Indicators

The Boolean controls and indicators (Fig) are used to enter and display Boolean (True or false) values. Boolean objects simulate switches, push buttons, and LEDs. The most common Boolean objects are the vertical toggle switch and the round LED.



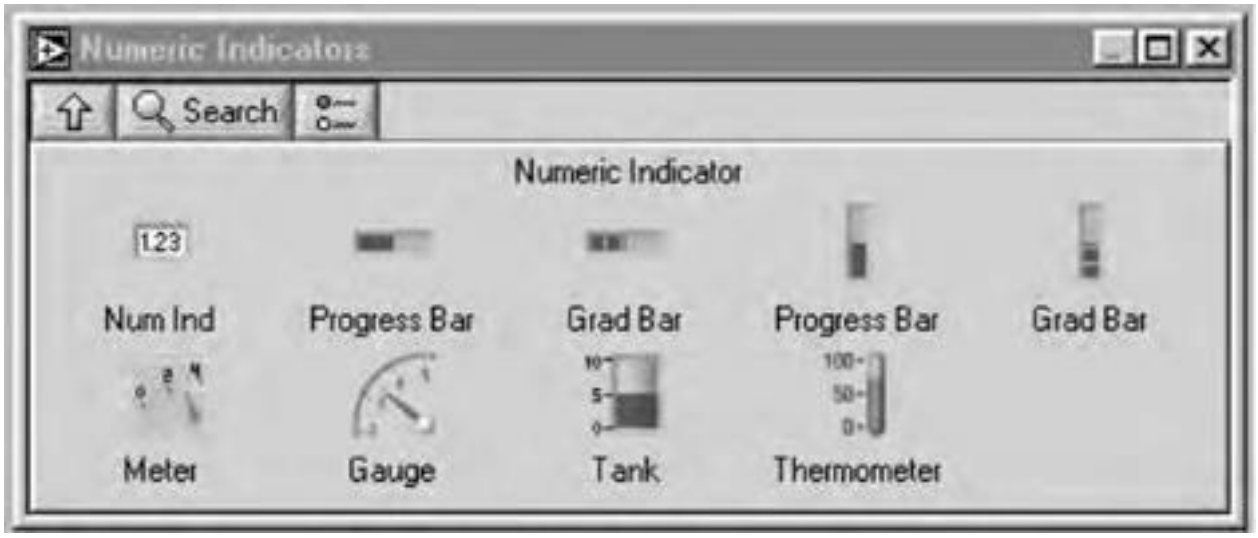


Fig 5: Numeric Control and Indicator

Block Diagram

The block diagram window holds the graphical source code of a LabVIEW'sblock diagram corresponds to the lines of text found in a more conventional language like C or BASIC – it is the actual executable code. The block diagram can be constructed with the basic blocks such as: terminals, nodes, and wires.

Terminals

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Terminals are analogous to parameters and constants in text-based programming languages.

Types of terminals include control or indicator terminals and node terminals. Control and indicator terminals belong to front panel controls and indicators. When a control or indicator is placed on the front panel, LabVIEW automatically creates a corresponding terminal on the block diagram. The terminals represent the data type of the control or indicator. The user cannot delete a block diagram terminal that belongs to a control or indicator. The terminal disappears when its corresponding control or indicator is deleted on the front panel. The front panel controls or indicators can be configured to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals.

Control terminals have thick borders, while indicator terminal borders are thin. It is very important to distinguish between the two since they are not functionally equivalent i.e., control=input, indicator=output, and so they are not interchangeable.

Nodes:

Nodes are analogous to statements, operators, functions, and subroutines in standard programming languages. The AND and XOR functions represent one type of node. A structure is another type of node. Structures can execute code repeatedly or conditionally, similar to loops and Case statements in traditional programming languages. LabVIEW also has special nodes, called Formula Nodes, which are useful for evaluating mathematical formulas or expressions.

Wires

Wires connecting nodes and terminals hold a LabVIEW VI together. Wiresare the data paths

between source and destination terminals, they deliver data from one source terminal to one or more destination terminals. Wires are analogous to variables in text-based programming languages. If more than one source or no source at all is connected to a wire, LabVIEW disagrees and the wire will appear broken. This principle of wires connecting source and destination terminals explains why controls and indicators are not interchangeable; controls are source terminals, whereas indicators are destinations, or sinks. Each wire has a different style or color, depending on the data type that flows through the wire. The below Table shows a few wires and their corresponding types. By default, basic wire styles are used in block diagrams. To avoid confusion among the data types, the colors and styles are simply matched.

Block Diagram Toolbar

When a VI is run, buttons appear on the block diagram toolbar that can beused to debug the VI.

Table 2: Data type and representation

	Scalar	1D array	2D array	Color
Floating point number				Orange
Integer number				Blue
Boolean				Green
String				Pink

Table 3 :Dialogue box menu

S No	Icon	Name	Meaning
1	New...	New	The **New** button is used to create a new VI. The arrow on the **New** button is used to open a blank VI or to open the **New** dialog box.
2	Open...	Open	The **Open** button is used to open an existing VI. The arrow on the **Open** button is used to open recent files.
3	Configure...	Configure	The **Configure** button is used to configure the data acquisition devices. The arrow on the **Configure** button is used to configure LabVIEW.
4	Help...	Help	The **Help** button is used to launch the *LabVIEW Help*. The arrow on the **Help** button is used for other Help options, including the NI Example Finder.

Startup Menu

The menus at the top of a VI window contain items common to other applications, such as Open, Save, Copy, and Paste, and other items specific to LabVIEW. Some menu items also list shortcut key combinations (Mac OS).The menus appear at the top of the screen (Windows and UNIX).The menus display only the most recently used items by default. The arrows at the bottom of a menu are used to display all items. All menu items can be displayed by default by selecting Tools->Options

Palletes

LabVIEW has graphical; floating palettes to help the user to create and run VIs.LabVIEW has three often-used floating palettes that can be placed in a convenient spot on the screen: the Tools palette, the Controls palette, and the Functions palette. Controls and Functions Palettes. The Controls palette will often be used, since the controls and indicators that are required on the front panel are available. The user will probably use the Functions palette even more often, since it contains the functions and structures used to build a VI. The Controls and Functions palettes are unique in several ways. Most importantly the Controls palette is only visible when the front panel window is active, and the Functions palette is only visible when the block diagram window is active. Both palettes have subpalettes containing the objects to be accessed. As the cursor is passed over each subpalette button in the Controls and Functions palettes, the subpalette's name appears at the top of the window. If a button is clicked, the associated subpalette appears and replaces the previous active palette. To select an object in the subpalette, the mouse buttonis clicked over the object, and then clicked on the front panel or block diagram to place it wherever desired. Like palette button names, subpalette object name appear when the cursor is run over them.

To return to the previous("owning") palette, the top-left arrow on each palette is selected. Clicking on the spyglass icon the user can search for a specific item in a palette, and then the user can edit palettes by clicking the options buttons. There is another way to navigate palettes that some people find a little easier. Instead of having each subpalette replace the current palette, the user can pass through subpalettes in a hierarchical manner without them replacing their parent palettes.

Controls Palette

The Controls palette can be displayed by selecting Window->Show ControlsPalette or right-clicking the front panel workspace. The Controlspalette can also be tacked down by clicking the thumbtack on the top left corner of the palette. By default, the Controls palette starts in the Expressview. The Express palette view includes subpalettes on the top level of theControls and Functions palettes. The All Controls and All Functions subpalettes contain the complete set of built-in controls, indicators, VIs, and functions. The Advanced palette view includes subpalettes on the top level of the Controls and Functions palettes that contain the complete set of built in controls, indicators, VIs, and functions. The Express subpalettes contain Express VIs and other objects required to build common measurement applications. Click the Options button on the Controls or Functions palette to change to another palette view or format.

Functions Palette

The Functions palette is available only on the block diagram. The Functions palette contains the VIs and functions used to build the block diagram. The Functions Palette can be displayed by selecting the Windows->Show or right-clicking the block diagram workspace to display the Functions palette.

Fig : 6 Function Palette

Tools Palette

A tool is a special operating mode of the mouse cursor. Tools are used to perform specific diting and operation functions, similar to that used in a standard paint program.

Fig :7 Tool Palette

Table 4 : Tool Pal

S. No	Icon	Name	Meaning
1		Operating Tool	The Operating Tool is used to change values of front panel controls and indicators. Used to operate knobs, switches, and other objects with the Operating tool – hence the name. It is the only front panel tool available when the VI is running or in run mode.
2		Positioning tool	The Positioning tool selects, moves, and resizes objects.
3		Labeling tool	The Labeling tool creates and edits text labels.
4		Wiring tool	The Wiring tool wires objects together on the block diagram. It is also used to assign controls and indicators on the front panel to terminals on the VI's connector.
5		Color tool	The Color tool brightens objects and background by allowing the user to choose from a multitude of hues. Both foreground and background colors can be selected by clicking on the appropriate color area in the Tools palette. If an object is popped up with the Color tool, a hue from the color palette appears and the required color can be chosen.

Loops & Charts:

LabVIEW offers two loop structures namely, the For Loop and While Loop to control repetitive operation in a VI. A For Loop executes a specific number of times; a While Loop executes until a specified condition is no longer true.

- **The FOR Loop**

A For Loop executes the code inside its borders, called its sub diagram, for total of count times, where the count equals the value contained in the count terminal. The count can be set by wiring a value from outside the loop of the count terminal. If '0' is wired to the count terminal, the loop does not execute.The iteration terminal contains the current number of completed loop iterations; 0 during the first iteration, 1 during the second, and so on, up to N-1 (where N is the number of times the loop executes).

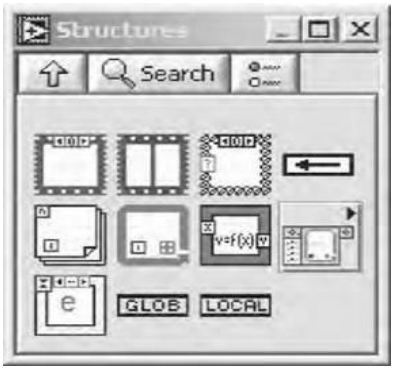


Fig 1: Structure Palette

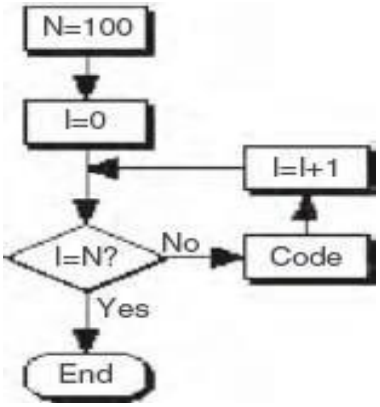


Fig 2: Flow chart of FOR loop

Fig 3: FOR Loop

The For Loop is located on the **Functions**->**All Functions**->**Structures**
The value in the count terminal (an input terminal), indicates how many times to repeat the sub diagram.
The iteration terminal (an output terminal), contains the number of iterations completed. The iteration count always starts at zero. During the first iteration, the iteration terminal returns 0.

- **The While Loop**

The While Loop executes the subdiagram inside its borders until the Boolean value wired to its conditional terminal is FALSE. LabVIEW checks the conditional terminal value at the end of iteration. If the value is TRUE, the iteration repeats. The default value of the conditional terminal is FALSE, so if left unwired, the loop iterates only once.

The While Loop is equivalent to the following pseudocode:

Do
Execute sub diagram
While condition is TRUE

It change the state that the conditional terminal of the While Loop checks, so that instead looping while true, we can have it loop unless it'strue. To do this, we pop-up on the conditional terminal, and select **"Stop ifTrue."**

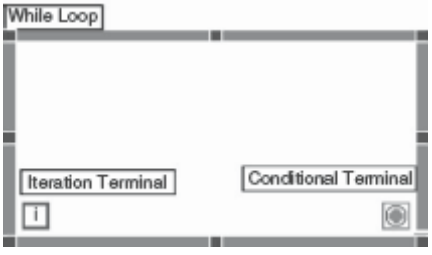


Fig 4: While loop terminals

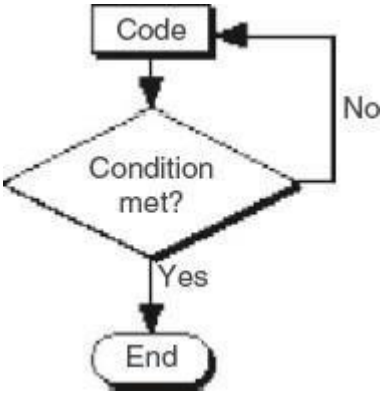


Fig 5: Flowchart of while loop

Do

Execute subdiagram

While condition is NOT TRUE

- The While Loop is located on the **Functions**>>**Execution Control** palette
- The section of code to be added inside the While loop is dragged or while loop encloses the area of code to be executed conditionally.

Condition Terminal

The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value. The default behavior and appearance of the conditional terminal is **Stop If True**. When a conditional terminal is **Stop If True**, the While Loop executes its subdiagram until the conditional terminal receives a True value.

Iteration Terminal

The iteration terminal, an output terminal, contains the number of completed iterations. The iteration count always starts at zero. During the first iteration, the iteration terminal returns 0.

EXAMPLE 1

Problem statement: To find the sum of first 10 natural numbers using For Loop.

Block diagram construction:

– The For Loop is added from the structures sub palette on the functions Palette located on the block diagram.
– The count terminal is wired with the value 10 for 10 natural numbers.
– The iteration terminal is wired the greater than or equal to node from the Comparison subpalette from the Functions palette.
– The Boolean Output of the greater than or equal to is wired to the conditional terminal as in Fig.

Front panel construction:

– The control and indicator are added from the controls palette of the front panel. Using the labeling tool the Owned label is changed as input and output, respectively.

– The front panel with the result is shown in Fig. .

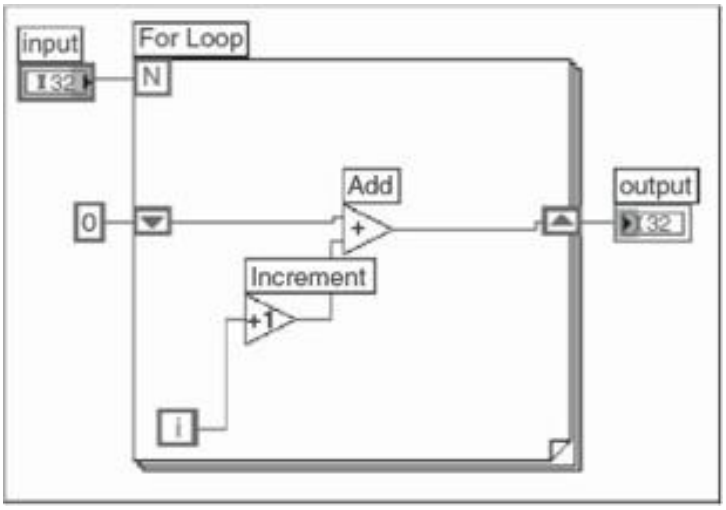


Fig 7: Block diagram of for loop example

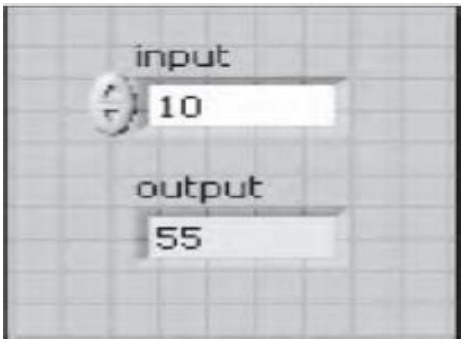


Fig 8: Front panel of For loop example

Example 2

Problem statement: To find the sum of first 10 natural numbers using while loop

Block diagram construction:

– The While Loop is added from the structures subpalette on the functions palette located on the block diagram.

– The count terminal is wired with the value 10 for 10 natural numbers.

– The iteration terminal is wired the greater than or equal to node from the comparison subpalette from the Functions palette.

– The Boolean Output of the greater than or equal to is wired to the conditional terminal of the While Loop as in Fig.

Front panel construction:

– The control and indicator are added from the controls palette of the front panel as in Fig. Using the labeling tool the Owned label is changed as input and output, respectively.

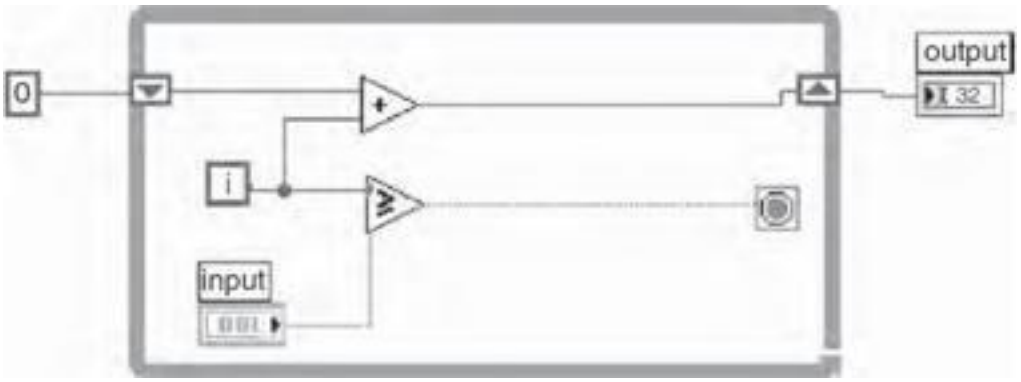


Fig 9: Block diagram of while loop for the above example

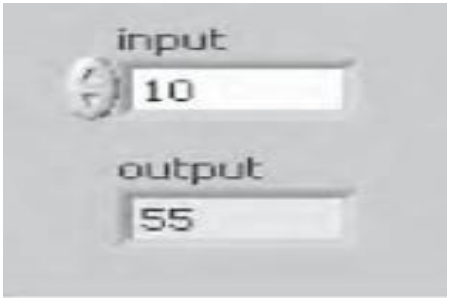


Fig 10: Front panel

Arrays:

A LabVIEW array is a collection of data elements that are all the same type similar to traditional programming languages. An array can have one or more dimensions, and has elements 2^{31}per dimension (memory permitting, of course). An array element can have any type except another array, a chart, or a graph.

Single and Multidimensional Arrays:

Single Dimensional Array:

Array elements are accessed by their indices; each element's index is in range 0 to N − 1, where N is the total number of elements in the array.The *first* element has index 0, the *second* element has index 1, and so on. Generally, waveforms are often stored in arrays, with each point in the waveform comprising an element of the array. Arrays are also useful for storing data generated in loops, where each loop iteration generates one element of the array.

Two steps are involved to make the controls and indicators for compound data types such as arrays and clusters.

Step 1. The array control or indicator known as the Array shell is created from the **Array** subpalette of the **Controls** palette available in the front panel
Step 2. The array shell is combined with a data object, which can be numeric, Boolean, or string (or cluster).Using the above two steps the created structure resembles

Index	0	1	2	3	4	5	6	7	8	9
10-element array	12	32	82	8.0	4.8	5.1	6.0	1.0	2.5	10

Fig11: Single Dimensional Array

Fig12: Array control and indicator

The user can also create array constants on the block diagram just like creating numeric, Boolean, or string constants. **Array Constant** can be chosen from the **Array** subpalette of the **Functions** palette. Then the data is placed in an appropriate data type (usually another constant) similar to that created on the front panel. This feature is useful when the user needs to initialize shift registers or provide a data type to a file or network functions. To clear an array control, indicator, or constant of data, the user can pop-on the index display.

Two-Dimensional Arrays

A two-dimensional, or 2D, array stores elements in a gridline fashion. It requires two indices to locate an element: a column index and a row index, both of which are zero-based in LabVIEW

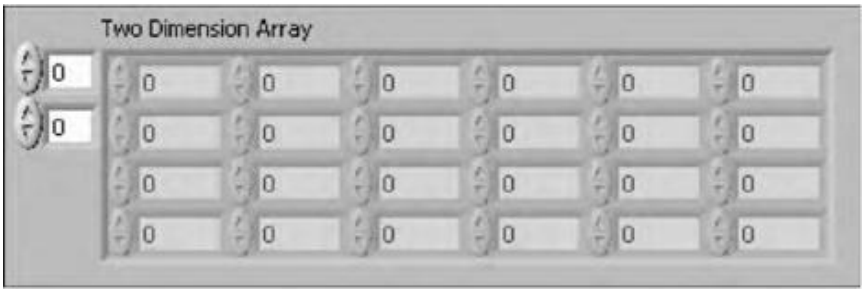


Fig13: 6*4 2D array

The user can add dimensions to an array control or indicator by popping up on its index display (not on the element display) and choosing **Add Dimension** from the Pop-up menu. In a 2D array of digital controls the user will have two indices to specify each element. The grid cursor of the Positioning tool can be used to expand the element display in two dimensions so that more elements can be seen. Unwanted dimensions can be removed by selecting **Remove Dimension** from the index display's Pop-up menu.

Creating Two-Dimensional Array

Two For Loops, one inside the other, can be used to create a 2D array on the front panel. The

inner For Loop creates a row, and the outer For Loop “stacks” these rows to fill in the columns of the matrix. In two For Loops creating a 2D array of random numbers using auto indexing

Autoindexing

The For Loop and the While Loop can index and accumulate arrays at their boundaries automatically, adding one new element for each loop iteration. This capability is called ***autoindexing***. One important thing to remember is that autoindexing enabled by default on For Loops but disabled by default on While Loops. The For Loop autoindexes, an array at its boundary. Each iteration creates the next array element. After the loop completes, the arraypasses out of the loop to the indicator; none of the array data are available until the Loop finishes. Notice that the wire becomes thicker as it changes to an array wire type at the Loop border.

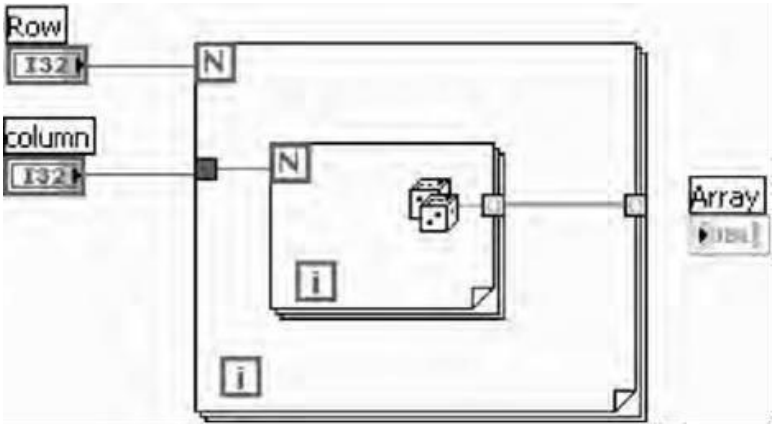


Fig14: For loop for creating 2 D array

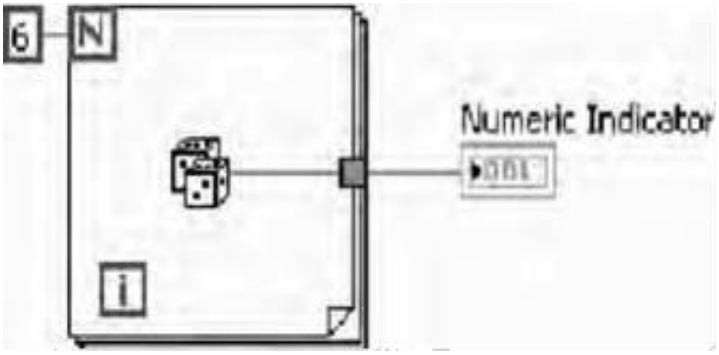


Fig15: Auto indexing disabled

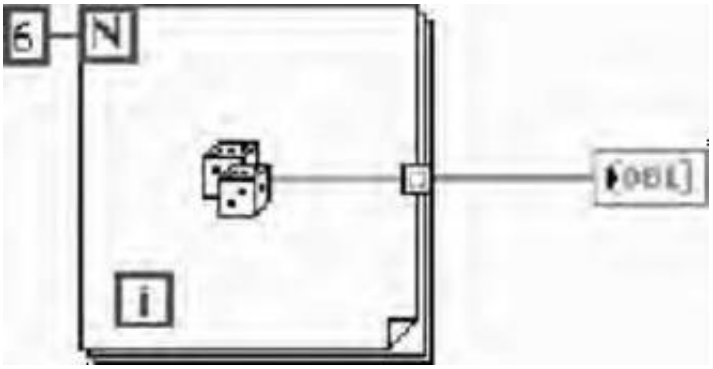


Fig16: Auto indexing enabled

To wire a scalar value out of a For Loop without creating an array, autoindexing must be disabled. This can be done by popping up on the tunnel (the square with the [] symbol) and choosing **Disable Indexing** from the tunnel's Pop-up menu. Since autoindexing is disabled by default, to wire array data out of a While Loop, the user must pop-up on the tunnel and select **Enable Indexing**. When autoindexing is disabled, only the last value returned from the **Random Number (0–1)** function passes out of the loop. Notice that the wire remains the same size after it leaves the loop. Pay attention to this wire size, because autoindexing is a common source of problems among beginners. Autoindexing also applies when the user is wiring arrays into loops. If indexing is enabled, the loop will index off one element from the array each time it iterates (note how the wire becomes thinner as it enters the loop). If indexing is disabled, the entire array passes in to the loop at once. Because For Loops are often used to process arrays, LabVIEW enables autoindexing by default when an array is wired into or out of them.

Clusters :

Cluster is a data structure that groups data. However, unlike an array, a cluster can group data of different types (i.e., numeric, Boolean, etc.); it is analogous to a struct in C or the data members of a class in C++ or Java.

Cluster elements can be accessed by *unbundling* them all at once or by indexing one at a time, depending on the function chosen; each method has its place. Unbundling a cluster is similar as unwrapping a telephone cable and having access to the different-colored wires. Unlike arrays, which can change size dramatically, clusters have a fixed size, or a fixed number of wires in them. The Unbundled By Name function is used to access specific cluster elements. Cluster terminals can be connected with a wire only if they have exactly the same type; in other words, both clusters must have the same number of elements, and corresponding elements must match in both data type and order. The principle of polymorphism applies to clusters as well as arrays, as long as the data types match.

Clusters are often used in error handling. The error clusters, **Error In.ctl** and **Error Out.ctl**, are used by LabVIEW to pass a record of errors between multiple VIs in a block diagram

Creating Cluster Controls and Indicators

A cluster is created by placing a **Cluster** shell (**Array & Cluster** subpalette of the **Controls** palette) on the front panel. Like arrays, objects can be added directly inside when they are pulled out from the **Controls** palette, or the user can drag an existing object into a cluster. **Objects inside a cluster mustbe all controls or all indicators**.

A cluster cannot have a combination of both controls and indicators; this is based on the status of the first object one place inside it. The cluster can be resized with the Positioning tool if necessary. The cluster can conform exactly to the size of the objects inside it, by popping up on the border(not inside the cluster) and choosing an option in the **Auto sizing** menu.

Creating Cluster Constants

If a cluster control or indicator is available on the front panel and if the user wants to create a cluster constant containing the same elements on the block diagram, then the user can either drag that cluster from the front panel to the block diagram or right-click the cluster on the front panel and select **Create**>>**Constant** from the shortcut menu.

Fig17: Cluster palette

Bundling Data

The **Bundle** function (**Cluster** palette) assembles individual components into a new cluster or allows us to replace elements in an existing cluster. The function appears as the icon at the left when one places it in the diagram window. Dragging a corner of the function with positioning tool can increase the number of inputs. When wired on each input terminal, a symbol representing the data type of the wired element appears on the empty terminal. The order of the resultant cluster will be the order of inputs to the **Bundle**. To create a new cluster the user need not wire an input to the center **cluster** input of the **Bundle** function. This needs to be wired only if elements are replaced in the cluster.

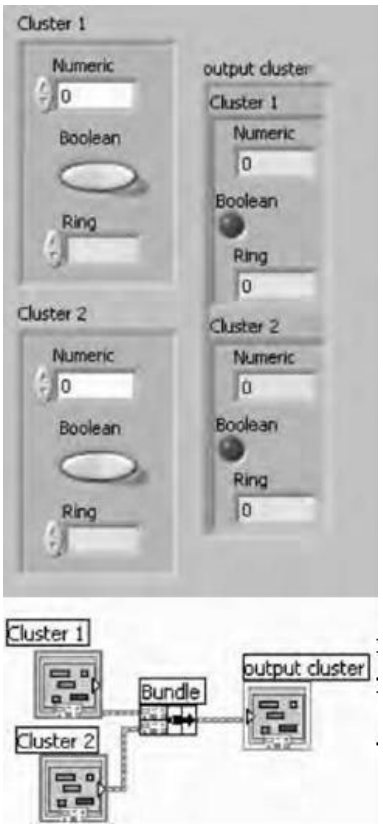


Fig18: Bundle Function

Unbundling Clusters

The **unbundled** function (**Cluster** palette) in Fig. 19 splits a cluster into each of its individual components. The output components are arranged fromtop to bottom in the same order they have in the cluster. If they have the same data type, the elements' order in the cluster is the only way to distinguish among them. Dragging a corner in the function with the Positioning tool can increase the number of outputs. The **Unbundled** function must be sized to contain the same number of outputs as there are elements in the input cluster, or it will produce bad wires. When an input cluster is wired to the correctly sized **Unbundled**, the previously blank output terminals will assume the symbols of the data types in the cluster example has been shown in Fig. LabVIEW does have a way to bundle and unbundled clusters using element names.

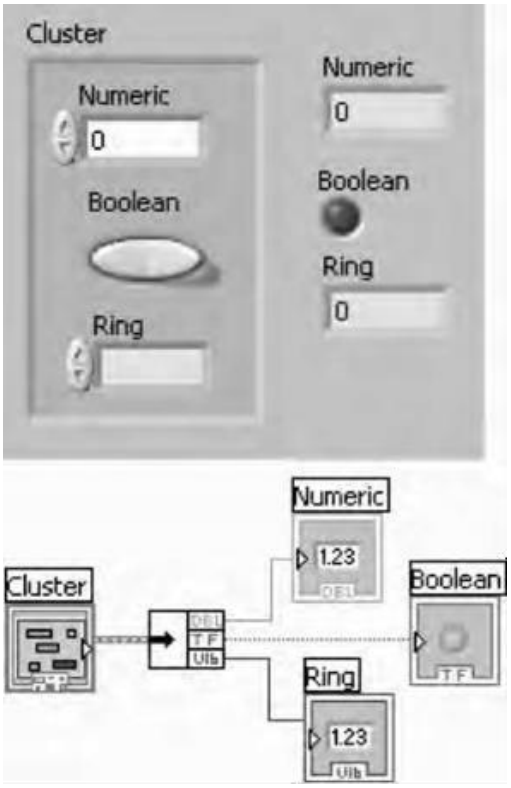


Fig19: Unbundle function

Bundling and Unbundling by Name

Sometimes there is no need to assemble or disassemble an entire cluster – the user just needs to operate on an element or two. This is accomplished using Bundle By Name and Unbundle By Name functions. Unbundle By Name, also located in the Cluster palette, returns elements whose name(s) are specified. There is no need to consider cluster order to correct Unbundle function size. The unbundle function is illustrated in Fig.

Bundle by Name, found in the Cluster palette, references elements by name instead of by position (as Bundle does). Unlike Bundle, we can access only the elements that are required. However, Bundle by Name cannot create new clusters; it can only replace an element in an existing cluster. Unlike Bundle, Bundle by Name's middle input terminal should be wired to allow the function know which element in the cluster has to be to replaced. This function is illustrated in Fig. All cluster elements should have owned labels when the By Name functions are used. As soon as

the cluster input of Bundle By Name or Unbundled By Name is wired, the name of the first element in the cluster appears in the name input or output.

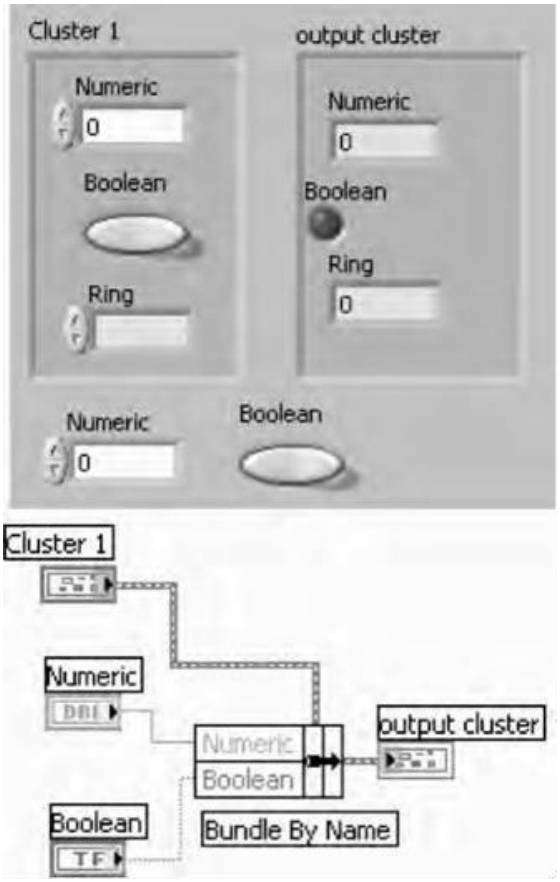


Fig20: Bundle by name

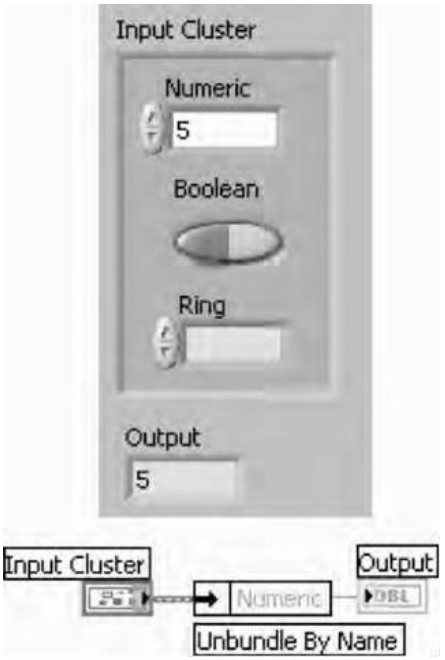


Fig21: Unbundle by name

Graphs:

The graphs located on the Controls_Graph Indicators palette include the waveform graph and XY graph. The waveform graph plots onlysingle-valued functions, as in $y = f(x)$, with points evenly distributed along the x-axis, such as acquired time-varying waveforms.

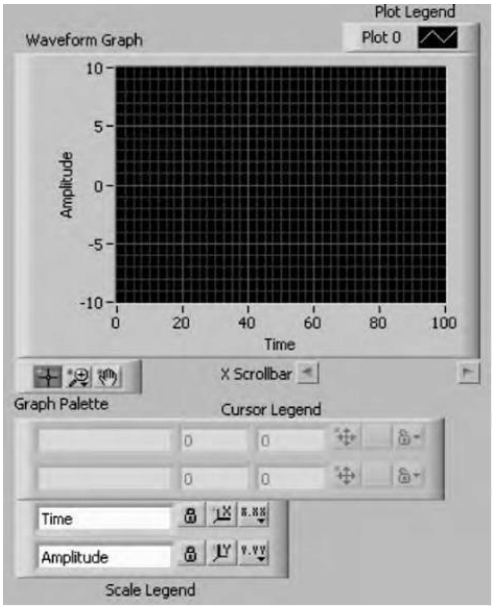


Fig 22: waveform graph

Single-Plot Waveform Graphs

The waveform graph accepts a single array of values and interprets the data as points on the graph and increments the x index by one starting at $x = 0$. The graph also accepts a cluster of an initial x value, and an array of y data.

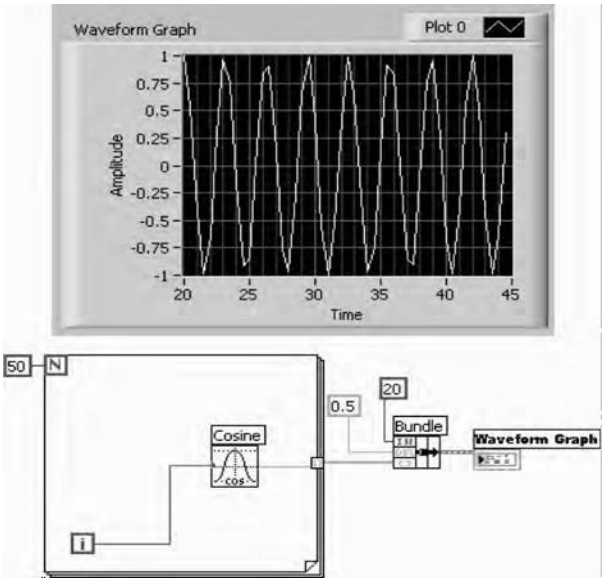


Fig 23: Single Plot Graph

Multiple-Plot Waveform Graphs

A multi plot waveform graph accepts a 2D array of values, where each row of the array is a single plot. The graph interprets the data as points on the graph and increments the *x* index by one, starting at $x = 0$. Wire a 2D array data type to the graph, right-click the graph, and select Transpose Array from the shortcut menu to handle each column of the array as a plot.

XY Graphs

XY graphs display any set of points, evenly sampled or not. Resize the plot legend to display multiple plots. Use multiple plots to save space on the front panel and to make comparisons between plots. XY and waveform graphs automatically adapt to multiple plots.

Single Plot XY Graphs

The single-plot XY graph accepts a cluster that contains an *x* array and a *y* array. The XY graph also accepts an array of points, where a point is a clusterthat contains an *x* value and a *y* value.

Multiplot XY Graphs

The multiplot XY graph accepts an array of plots, where a plot is a cluster that contains an *x* array and a *y* array. The multi plot XY graph also accepts an array of clusters of plots, where a plot is an array of points.

Waveform Charts

A plot is simply a graphical display of *X* versus *Y* values. Often, *Y* values in a plot represent the data value, while *X* values represent time. The waveform chart, located in the Graph subpalette of the Controls palette, is a special numeric indicator that can display one or more plots of data.

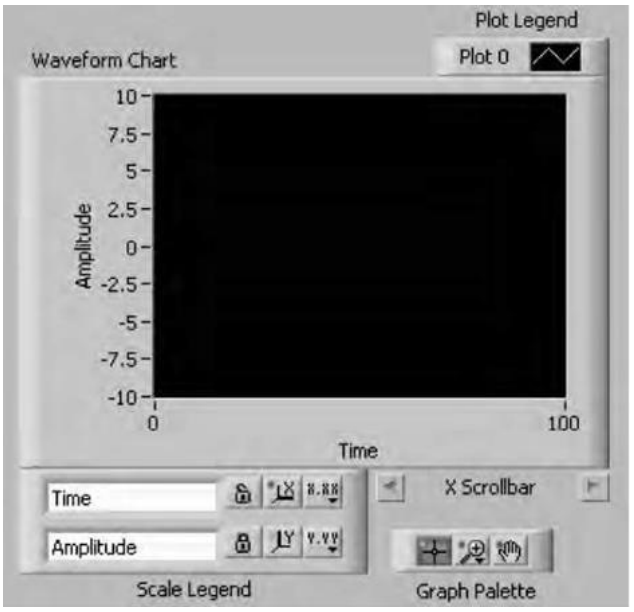


Fig24: Waveform chart and its component

Chart Update Modes

The waveform chart has three update modes – strip chart mode, scope chart mode, and sweep

chart mode. The update mode can be changed by poppingup on the waveform chart and choosing one of the options from the **Advanced**>>**update Mode**>>menu. Modes can be changed while the VI isrunning by selecting **Update Mode** from the chart's runtime Pop-up menu.The default mode is **Strip Chart**. The strip chart has a scrolling display similar to a paper strip chart . The scope chart and the sweep chart have retracing displays similar to that of an oscilloscope.

Strip Chart

A strip chart shows running data continuously scrolling from left to right across the chart.

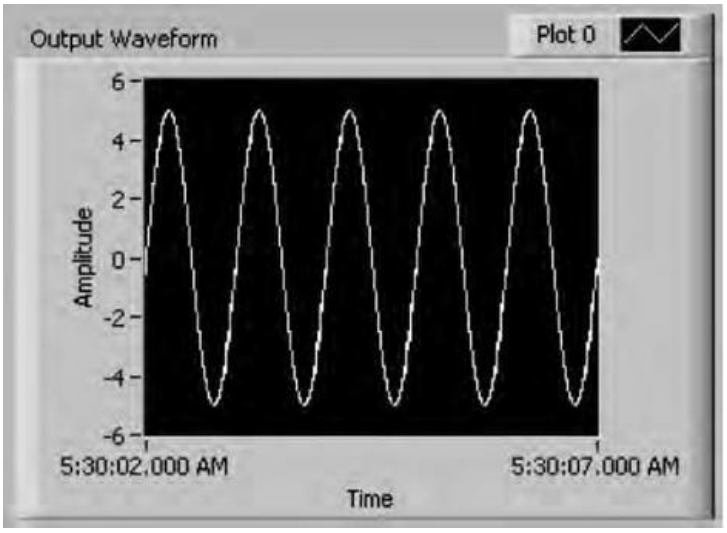


Fig:25 Strip chart

Scope chart:

A scope chart shows one item of data, such as a pulse or wave, scrolling part way across the chart

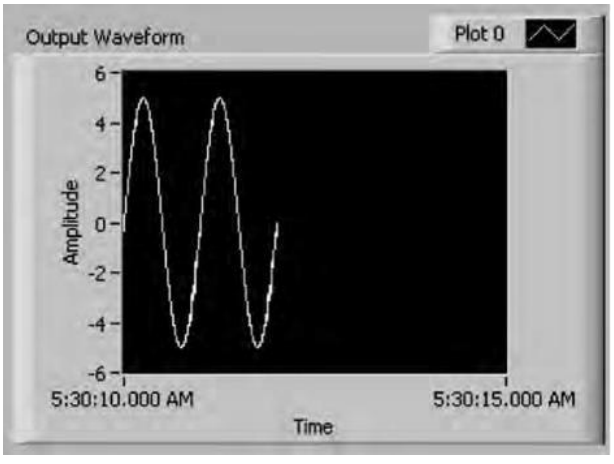


Fig: 26 Scope chart

Sweep Chart:

A sweep chart is similar to an EKG display. A sweep chart works similarly to a scope except it shows the older data on the right and the newer data on the left separated by a vertical line. The scope chart and sweep chart have retracing displays similar to an oscilloscope. Because there is less overhead in retracing a plot, the scope chart and the sweep chart display plots significantly faster than the strip chart.

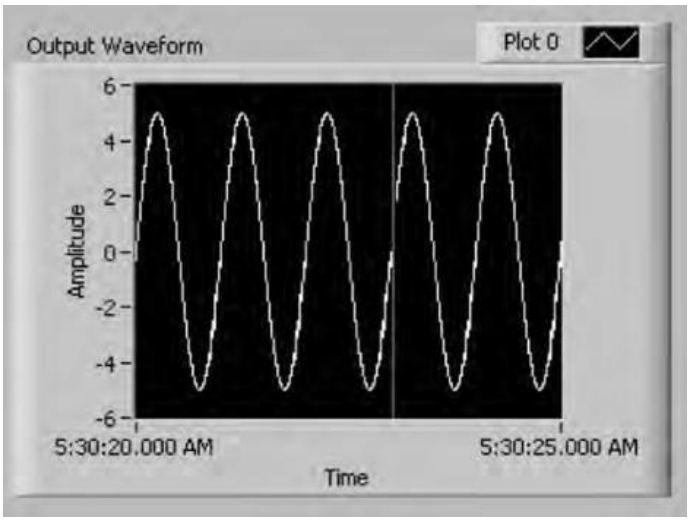


Fig:27 Sweep Chart

Wiring Charts

A scalar output can be wired directly to a waveform chart.

Strings

A string is a sequential collection of displayable or nondisplayable ASCII characters. Strings provide a platform-independent format for information and data. Often, strings may be used for displaying simple text message.

Some of the more common applications of strings include the following:

- Creating simple text messages.
- Passing numeric data as character strings to instruments and then converting the strings to numeric values.
- Storing numeric data to disk. To store numeric values in an ASCII file, the numeric values must be first converted to strings before writing the numeric values to a disk file.
- Instructing or prompting the user with dialog boxes. On the front panel, strings appear as tables, text entry boxes, and labels.

Creating String Controls and Indicators

The string control and indicator located on the Controls_Text Controls and Controls_Text Indicators palettes are used to simulate text entry boxes and labels. Using the Operating tool or labeling tool text data can betyped or edited in a string control. The Positioning tool is used to resize a frontpanel string object. The space occupied by a string object can be minimized byright-clicking the object and selecting the Visible Items_Scrollbar optionfrom the shortcut menu. The display types can be selected by right-clickinga string control or indicator on the front panel

String Functions

String Length returns the number of characters in a given string asshown in Fig 28.

Concatenate Strings concatenates all input strings into a single outputstring as shown in Fig 29.The function appears as the icon shown in block diagram in Fig 29.The function can be resized with the Positioning tool to increase the numberof inputs. In addition to simple strings, the user can also wire a 1D array ofstrings as input; the output will be a single string containing a concatenationof strings in the array.

String Subset accesses a particular section of a string. It returns the substringbeginning at offset and containing length which indicates the number

Fig:28 string length

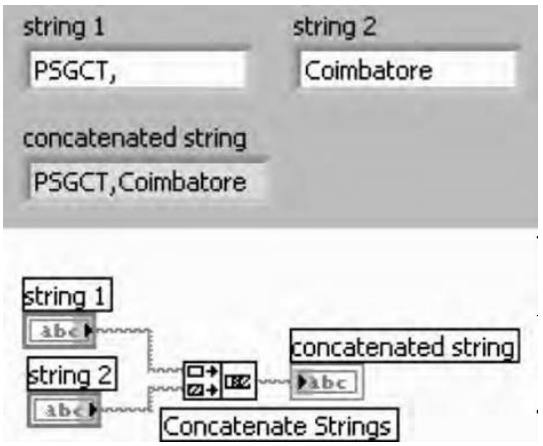


Fig 29: Concatenate Strings

Sequence Structures (Flat and Stacked Structures)

Determining the execution order of a program by arranging its elements ina certain sequence is called control flow. Visual Basic, C, and most other Fig. 30.

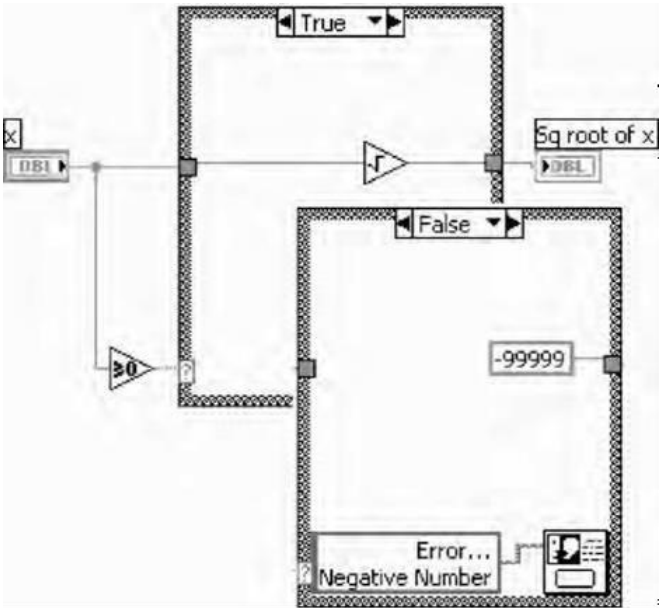


Fig30: Block diagram

Fig 31: Front panel

procedural programming languages have inherent control flow because statementsexecutes in the order in which they appear in the program. LabVIEWuses the *Sequence Structure* to obtain control flow within a dataflow framework.A Sequence Structure executes frame 0, followed by frame 1, then frame2, until the last frame executes. Only when the last frame completes dataleaves the structure.The Sequence Structure as shown in Fig. 30, looks like a frame of film.It can be found in the **Structures** subpalette of the **Functions** palette.Like the Case Structure, only one frame is visible at a time – the arrowsat the top of the structure can be selected to see other frames; or the topdisplay can be clicked to obtain a listing of existing frames, or the user canpop-up on the structure border and choose **Show Frame**. When a SequenceStructure is first dropped on the block diagram, it has only one frame; thus,it has no arrows or numbers at the top of the structure to designate whichframe is showing. New frames can be created by popping up on the structureborder and selecting **Add Frame After** or **Add Frame Before** as shownin Fig. 31.The Sequence Structure is used to control the order of execution of nodesthat are not data dependent on each other. Within each frame, as in the rest ofthe block diagram, data dependency determines the execution order of nodes.

The Formula Node

The *Formula Node* is a resizable box that is used to enter algebraic formulasdirectly into the block

diagram. The Formula Node is a convenient text-basednode used to perform mathematical

The case structureis LabVIEW's method of executing conditional text, sort of like an"if–then–else"statement. It is located in the **Structures** subpalette of the **Functions** palette. The Case Structure, has two or more subdiagrams, orcases; only one of them executes, depending on the value of the Boolean,numeric, or string value wired to the *selector terminal*.If a Boolean value is wired to the selector terminal, the structure has twocases, FALSE and TRUE. If a numeric or string data type is wired to the selector, the structure can have from zero to almost unlimited cases. Initially only two cases are available, but number of cases can be easily added. More than one value can be specified for a case, separated by commas. In addition, the user can always select a "Default" case that will execute if the value wired to the selector terminal doesn't match any of the other cases. When a case structure is first placed on the panel, the Case Structure appears in its Boolean form; it assumes numeric values as soon as a numeric data type is wired to its selector terminal.Case Structures can have multiple subdiagrams, but the user can see only one case at a time, sort of like a stacked deck of cards. Clicking on the decrement(left) or increment (right) arrow at the top of the structure displays the previous or next subdiagram, respectively. The user can also click on the display at the top of the structure for a pull-down menu listing all cases, and then pop-up on the structure border and select **Show Case**. If a floating-point number is wired to the selector, LabVIEW rounds that number to the nearest integer value. LabVIEW coerces negative numbers to 0 and reduces any value higher than the highest-numbered case to equal the number of that case. The selector terminal can be positioned anywhere along the left border. If the data type wired to the selector is changed from a numeric to a Boolean, cases 0 to 1 change to FALSE and TRUE. If other cases exist (2 to n), Lab-VIEW does not discard them, in case the change in data type is accidental. However, these extra cases must be deleted before the structure can execute. For string data types wired to case selectors, the user should always specify the case values as strings between quotes.

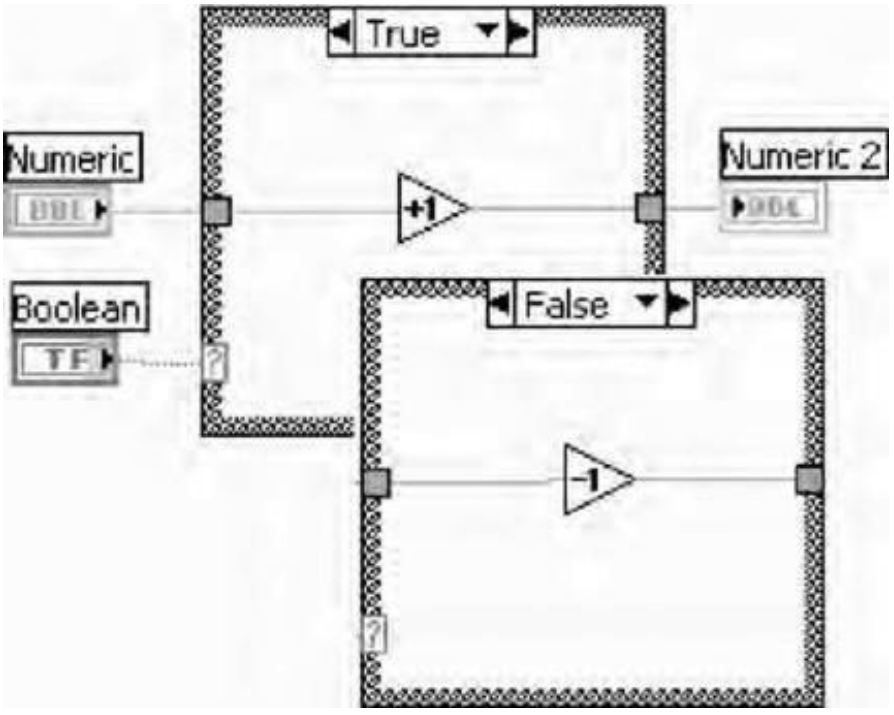


Fig 34: Boolean case structure

Boolean Case Structure

The following example is a Boolean Case structure shown in Fig34. The cases are shown overlapped to simplify the illustration. If the Boolean controlwired to the selector terminal is True, the VI increments the numeric value. Otherwise, the VI decrements the numeric value.

Integer Case Structure

The following example is an integer Case structure shown in Fig35. **Integer** is a text ring control located on the **Controls_Text Controls** palette that associates numeric values with text items. If the text ring control wired to the selector terminal is 0 (add), the VI decrements the numeric values. If the value is 1 (subtract), the VI increments the numeric values. If the text ring control is any other value than 0 (add) or 1 (subtract), the VI adds the numeric values, because that is the default case.

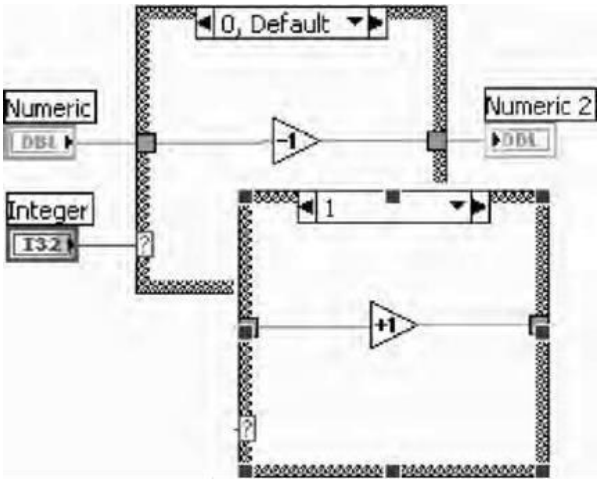


Fig 35: Integer Case Structure

String Case Structure

The following example is a string Case structure as shown in Fig36. If**String** is “Increment,” the VI increments the numeric values. If **String** is “Decrement,” the VI decrements the numeric values.

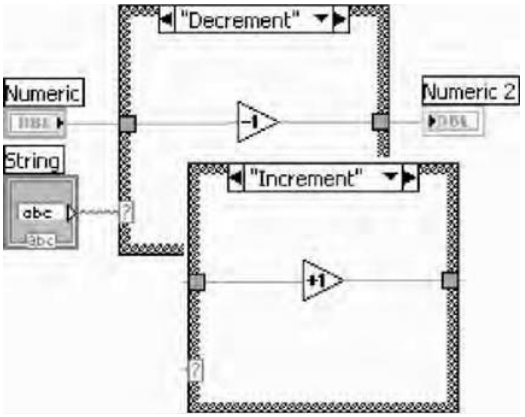


Fig 36: string case

Enumerated Case Structure

The following example is an enumerated Case structure as shown in Fig.37 An enumerated control gives users a list of items from which to select.

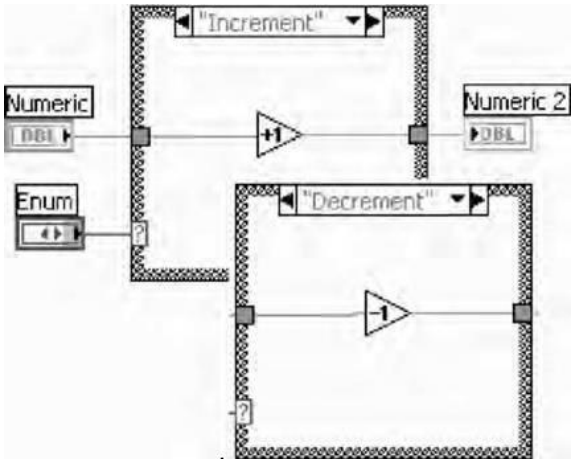


Fig:37 Enumerated case

The data type of an enumerated control includes information about the numeric values and string labels in the control. When an enumerated control is wired to the selector terminal of a Case structure, the case selector displays acase for each item in the enumerated control. The Case structure executes the appropriate case subdiagram based on the current item in the enumerated control. If Enum is "Increment," the VI increments the numeric values. If Enum is "Decrement," the VI decrements the numeric values.

Error Case Structure

When an error cluster is wired to the selector terminal of a Case structure, the case selector label displays two cases, Error and No Error, and the border of the Case structure changes color – red for Error and green for No Error. The Case structure executes the appropriate case subdiagram based on the error state. When an error cluster is wired to the selection terminal, the Case structure recognizes only the **status** Boolean of the cluster.

SEQUENCE STRUCTURES

A sequence structure contains one or more subdiagrams, or frames, that execute in sequential order. Within each frame of a sequence structure, as in the rest of the block diagram, data Dependency determines the execution order of nodes. There are two types of sequence structure the Flat Sequence structure and the Stacked Sequence structure. The Flat Sequence structure, shown as follows, displays all the frames at once and executes the frames from left to right and when all data values wired to a frame are available, until the last frame executes. The data values leave each frame as the frame finishes executing. Use the Flat Sequence structure to avoid using sequence locals and to better document the block diagram. When you add or delete frames in a Flat Sequence structure, the structure resizes automatically. To convert a Flat Sequence structure to a Stacked Sequence structure, right-click the Flat Sequence structure and select Replace with Stacked Sequence from the shortcut

menu. If you change a Flat Sequence to a Stacked Sequence and then back to a Flat Sequence,

LabVIEW moves all input terminals to the first frame of the sequence. The final Flat Sequence should operate the same as the Stacked Sequence. After you change the Stacked Sequence to a Flat Sequence with all input terminals on the first frame, you can move wires to where they were located in the original Flat Sequence.

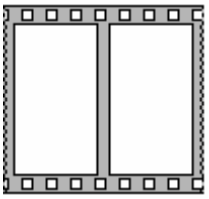


Fig:38 Flat sequence

Stacked Sequence Structure

The Stacked Sequence structure, shown as follows, stacks each frame so you see only one frame at a time and executes frame 0, then frame 1, and soon until the last frame executes. The Stacked Sequence structure returns data only after the last frame executes. Use the Stacked Sequence structure if you want to conserve space
on the block diagram. To convert a Stacked Sequence structure to a Flat Sequence structure, right-click the Stacked Sequence structure and select Replace»Replace with Flat Sequence from the shortcut menu. The sequence selector identifier, shown as follows, at the top of the Stacked Sequence structure contains the
current frame number and range of frames. Use the sequence selector identifier to navigate through the available frames and rearrange frames. The frame label in a Stacked Sequence structure is similar to the case selector label of the Case structure. The frame label contains the frame number in the center and decrement and increment arrows on each side. Click the decrement and increment arrows to scroll through the available frames. You also can click the down arrow next to the frame number and select a frame from thepull-down menu. Right-click the border of a frame, select Make This Frame, and select a framenumber from the shortcut menu to rearrange the order of a Stacked Sequence structure. Unlike the case selector label, you cannot enter values in the frame label. When you add, delete, or rearrange frames in a Stacked Sequence structure, LabVIEW automatically adjusts the numbers in the frame labels. To pass data from one frame to any subsequent frame of a Stacked Sequence structure, use a sequence local terminal shown .An outward-pointing arrow appears in the sequence local terminal of the frame that contains the data source. The terminal in subsequent frames contains an inward-pointing arrow, indicating that the terminal is a data source for that frame. You cannot use the sequence local terminal in frames that precede the first frame where you wired the sequence local.

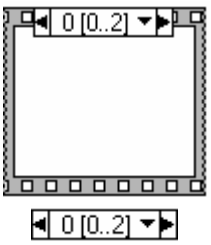

0 [0..2]
0 [0..2]

Fig 39: Stacked Structure

BASICS OF FILE INPUT/OUTPUT

File I/O records or reads data in a file. File I/O operations pass data to and from files. Use the fileI/ O VIs and functions located on the Functions->All Functions»File I/O palette to handle all aspects of file I/O, including the following:

- Opening and closing data files
- Reading data from and writing data to files
- Reading from and writing to spreadsheet-formatted files
- Moving and renaming files and directories
- Changing file characteristics
- Creating, modifying and reading configuration files

LabVIEW can use or create the following file formats: Binary, ASCII, LVM, and TDM.

- **Binary**—Binary files are the underlying file format of all other file formats.
- **ASCII**—An ASCII file is a specific type of binary file that is a standard used by most programs. It consists of a series of ASCII codes. ASCII files are also called text files.
- **LVM**—The LabVIEW measurement data file (.lvm) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The .lvm file includes information about the data, such as the date and time the data was generated. This file format is a specific type of ASCII file created for LabVIEW.
- **TDM**—This file format is a specific type of binary file created for National Instruments products. It actually consists of two separate files: an XML section contains the data attributes and a binary file for the waveform.

Use of Text Files

Use text format files for your data to make it available to other users or applications, if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important. Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file.

Use of Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storingany number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytesfor each number. Use binary files to save numeric data and to access specific numbers from a fileor randomly access numbers from a file. Binary files are machine readable only, unlike text fileswhich are human readable. Binary files are the most compact and fastest format for storing data.

Use of Datalog Files

Use datalog files to access and manipulate data only in LabVIEW and to store complex data structures quickly and easily. A datalog file stores data as a sequence of identically structured records, similar to a spreadsheet, where each row represents a record. Each record in a datalog File must have the same data types associated with it. LabVIEW writes each record to the file as acluster containing the data to store.

A typical file I/O operation involves the following process,
1. Create or open a file. Indicate where an existing file resides or where you want to create a new file by specifying a path or responding to a dialog box to direct LabVIEW to the file location. After

the file opens, a refnum represents the file.
2. Read from or write to the file.
3. Close the file.
File I/O VIs and some File I/O functions, such as the Read from Text File and Write to Text File functions, can perform all three steps for common file I/O operations. The VIs and functions designed for multiple operations might not be as efficient as the functions configured or designed for individual operations.

LOCAL VARIABLES

- Local variables transfer data within a single VI and allow data to be passed between parallel loops
- They also break the dataflow programming paradigm.

Two ways to create
a local variable are right-click on an object's terminal and select Create->Local Variable.

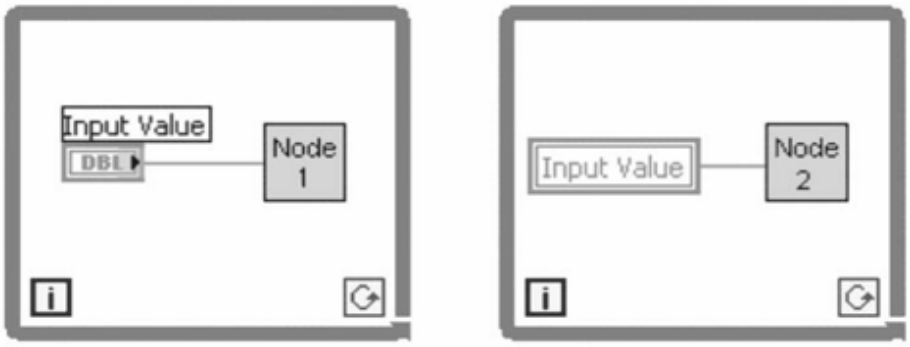


Fig 40: Local Variable

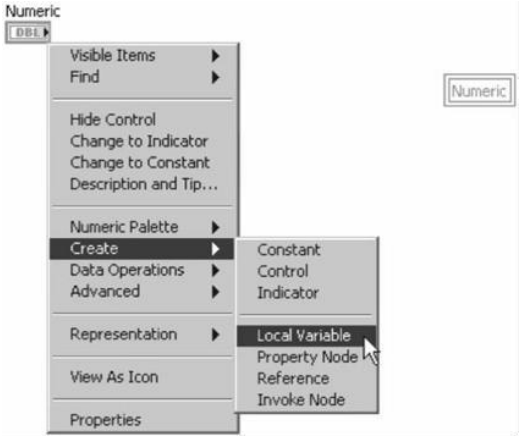


Fig41: Creating local variable

Another way is to select the Local Variable from the Structures palette. Create the front panel and

select a local variable from the Functions palette and place it on the block diagram. The local variable node, shown as follows, is not yet associated with a control or indicator. To associatea local variable with a control or indicator, right-click the local variable node and select Select Item from the shortcut menu.

GLOBAL VARIABLES

- Global variables are built-in LabVIEW objects.

use variables to access and pass data among several VIs that run simultaneously

- A local variable shares data within a VI; a global variable also shares data, but it shares data with multiple VIs.

For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You can use a global variable to terminate both loops with a single Boolean control as shown in Figure . If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

When you create a global variable, LabVIEW automatically creates a special global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables. Select a global variable as shown inFigure42 .from the *Functions* palette and place it on the block diagram. Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel. LabVIEW uses owned labels to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

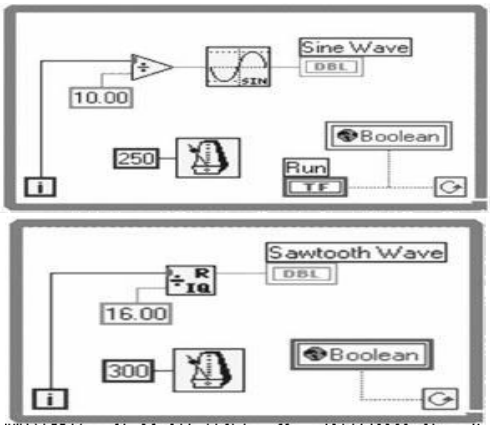


Fig:42 Global Variable

References:

1. Jerome, Virtual Instrumentation Using LabView, PHI, 2010
2. Lisa K. Wells and Jeffrey Travis, LABVIEW for Everyone, PHI, 1997.

3. Skolkoff, Basic concepts of LABVIEW 4, PHI, 1

Part-A

1. Define Virtual Instrumentation.
2. .What is Graphical System Design model?
3. Draw the VI and GSD model.
4. Draw the architecture of VI and indicate the parts.
5. Distinguish between Virtual Instrument and Traditional Instrument.
6. Mention the role of hardware's in VI I/O modules:
7. Mention the role of software's in VI
8. Name the different layers of VI software.
9. Mention the different challenges in Test.
10. What is G programming?

Part- B

1) Draw and explain the graphical and VI models with design flow
2) Explain the essential need for Virtual Instrumentation and compare it with the traditional instruments
3) Explain the role of different hardware's and software's in VI
4) Explain how VI can be used in test, Control and Design process
5) .Compare Graphical programming with traditional programming