LABVIEW

A Tutorial By

MASOOD EJAZ

Note: This tutorial is a work in progress and written specially for *CET 3464 – Software Applications in Engineering Technology*, a course offered as part of BSECET program at Valencia College. Any feedback will be much appreciated. This tutorial is based on LabVIEW 2010 version 10 that we have at Valencia College but can be used for any other version.

LabVIEW was developed to make it easier to collect data from laboratory instruments using data acquisition systems and further to process that data to yield important aspects of that data. Now, it has evolved into an extremely useful engineering software that can solve problems from a number of engineering fields. It can not only be used as a data acquisition and processing software but also to control different instruments and equipment.

LabVIEW interface is very graphical as against *MATLAB* where you have to do a lot of procedural programming. There is a plethora of Mathematical applications that you can solve using LabVIEW, including simple arithmetic applications, complex integration and differentiation, plots and graphs, statistics, and many more. It can be used to solve circuit problems, robotics problems, control problems, signal processing problems and many other problems from different engineering fields.

LabVIEW programs are called VIs, originally stood for *virtual instrument*, but LabVIEW is now used for many more applications than just creating a computer simulation if an instrument, and LabVIEW programs are typically referred to simply as *VI*s

Startup:

When you execute LabVIEW, title screen will show up informing you that LabVIEW is loading its processes and initializing. Once it is done loading its components, *Getting Started* screen will pop-up as shown in *figure 1*.

To create a LabVIEW program or LabVIEW *VI*, you have to click on *Blank VI* . Once you click it, *three* windows will pop-up; *Function Palette* associated with *Block Diagram*, and *Front Panel. Fornt Panel* displays the *controls* (knobs, buttons, graphs, etc.) and represents the graphical interface for the VI, whereas, *Block Diagram* holds the programming elements, called *blocks, functions,* or sometimes *subVIs*, that are wired together to build the graphical program.

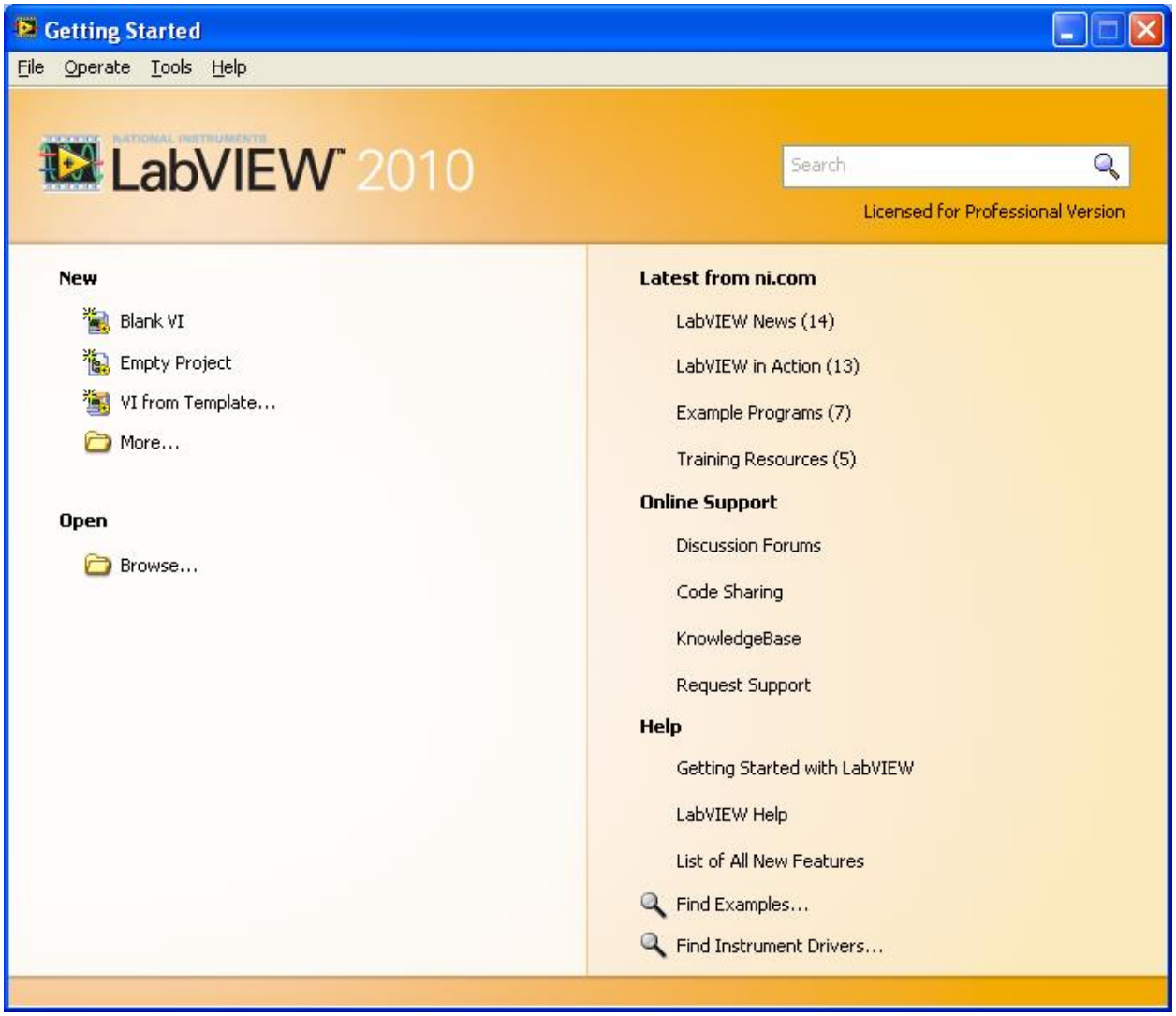


Figure 1: LabVIEW 'Getting Started' Screen

Palettes

Different palettes associated with LabVIEW are,

- Function Palette
- Control Palette
- Tool Palette

Function palette is associated with Block Diagram window, *Control palette* is associated with Front Panel Window, and *Tool Palette* is associated with both of the windows. All of the three palettes can be opened from *View* located in the menu bar.

Control palette provides access to the objects like controls, indicators, knobs, and graphs that are placed on the front panel. There is a large number of different controls available collected into a number of categories and each category can be expanded or collapsed.

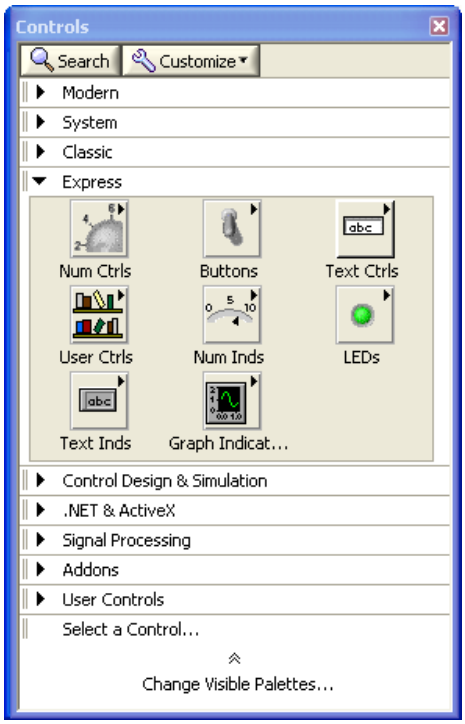


Figure 2: Control Palette Window

Function palette contains *functions*, *VIs*, and *Express VIs* that can be placed on a block diagram to create a graphical program. Like control palette, there is a large number ofdifferent functions available in the function palette collected into a number of different categories that can be expanded or collapsed

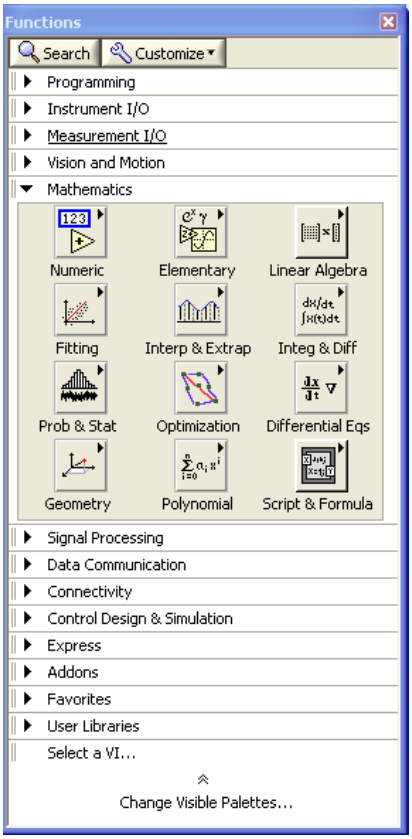


Figure 3: Function Palette Window

EXERCISE – 1

From *function palette* find the group(s) containing the following functions:

(i) Add

(ii) Wait (look for a wristwatch icon)

(iii) Transforms (Laplace, z, Fourier etc.)

From *control palette* find the group(s) containing:

(i) Dial Numeric Control

(ii) Toggle Switch

(iii) Table

Tool Palette is used to edit text, position/size/select, probe data and other jobs that will be learned and used as they will come across.

Figure 4: Tool Palette Window

Creating a VI

Let's start with a simple program. Choose a *toggle switch* and an *LED* from the control palette and put them on the *Front Panel* as shown in *figure 5*. Double click on the names of these controls and change them to *Power Switch* and *LED*

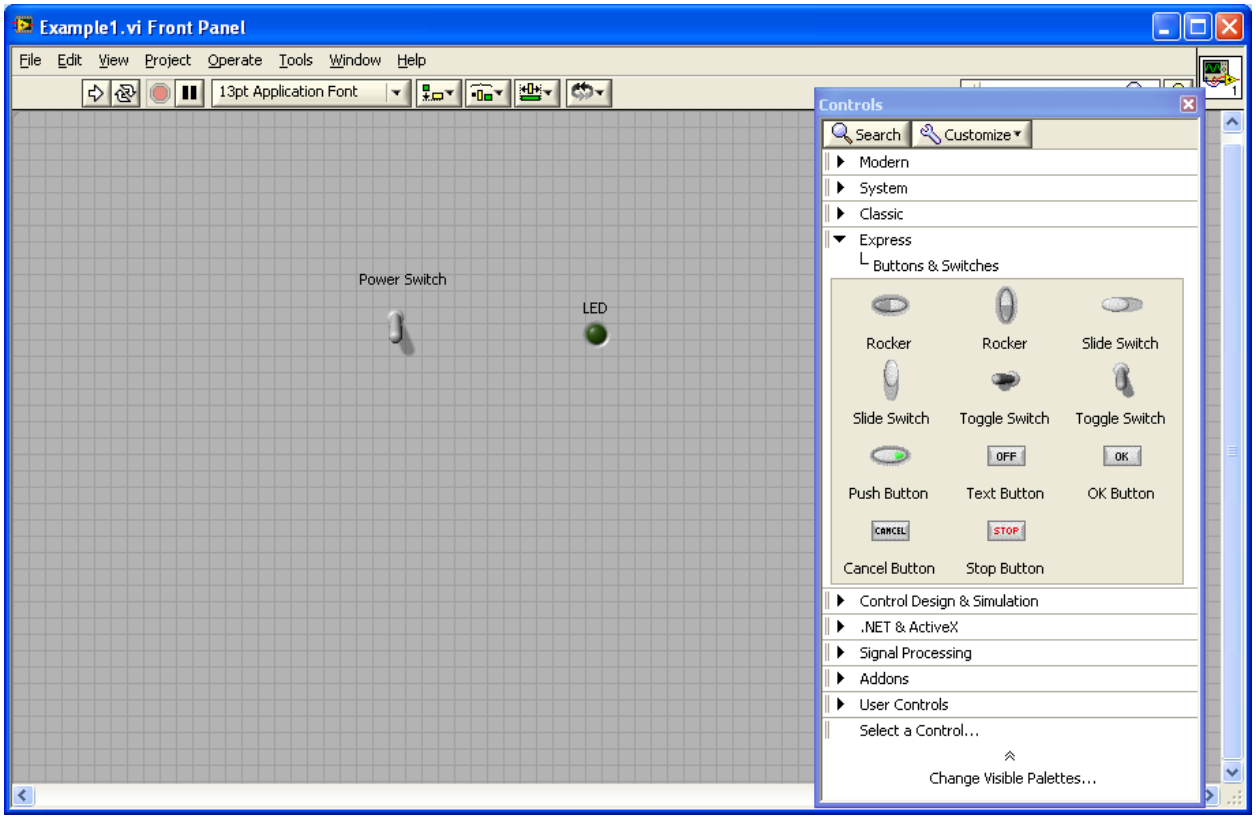


Figure 5: Power Switch and LED on the Front Panel

Now go to the *Block Diagram* window, functional blocks of your components will already be there. You will connect them in the block diagram through wire(s). You can wire different components in two ways; put your mouse on one of the connection end of any component and drag it to the connection end of any other component, or just click on the connection end of one of the components and click on the connection end of the other component.

NOTE: If you do not see your mouse as a *cross-hair* in your block diagram window, you are not in the *wiring mode*. Open *tool palette* and select either *Automatic Tool Selection* (one on the top) or *Connect Wire* tool. It will be highly recommended that you select *Automatic Tool Selection*.

When wiring is done, you are ready to simulate your design. Simulation is done in the *front panel* window. Simulation or *Run* buttons are located in the top toolbar menu and can be recognized as shown in *figure 6*

Figure 6: Run Buttons

From left to right, first button is to run your simulation once, second one is to run it continuously, third one is to stop the simulation, and fourth one is to pause the simulation. Run your simulation continuously and toggle the switch to see your LED turns ON and OFF.

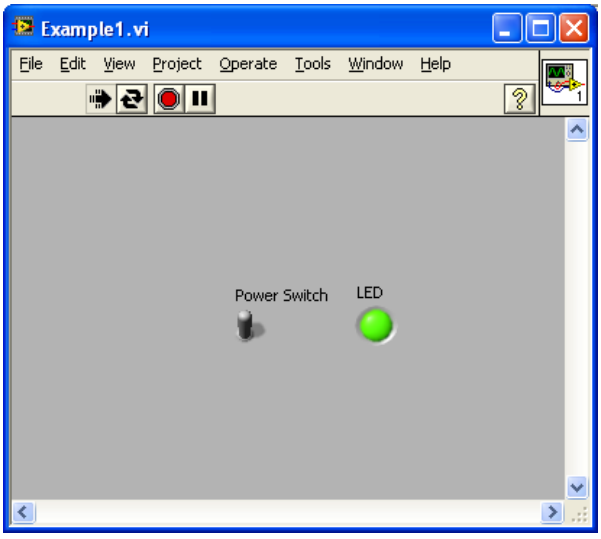


Figure 7: LED Operated by Switch

There are different mechanical properties for the switch that you can select. Select toggle switch, right-click your mouse and go to *Mechanical Action*. Try different types of mechanical actions for the switch.

You can also change color of your LED by selecting LED, going into its properties (right-click), and from there going to *Appearance* tab.

EXERCISE – 2

Instead of a toggle switch, use a *push-button momentary* switch and operate your LED through it. Change mechanical properties of switch to try different actions.

Mathematical Operations

Many basic and complex mathematical functions can be done with LabVIEW easily. Let's create a VI to add two numbers. Open a blank VI and choose *Num Ctrl* from *Numeric Controls* group. Place two numeric controls (*Addend* and *Augend)* in the front panel as shown in *figure 8*

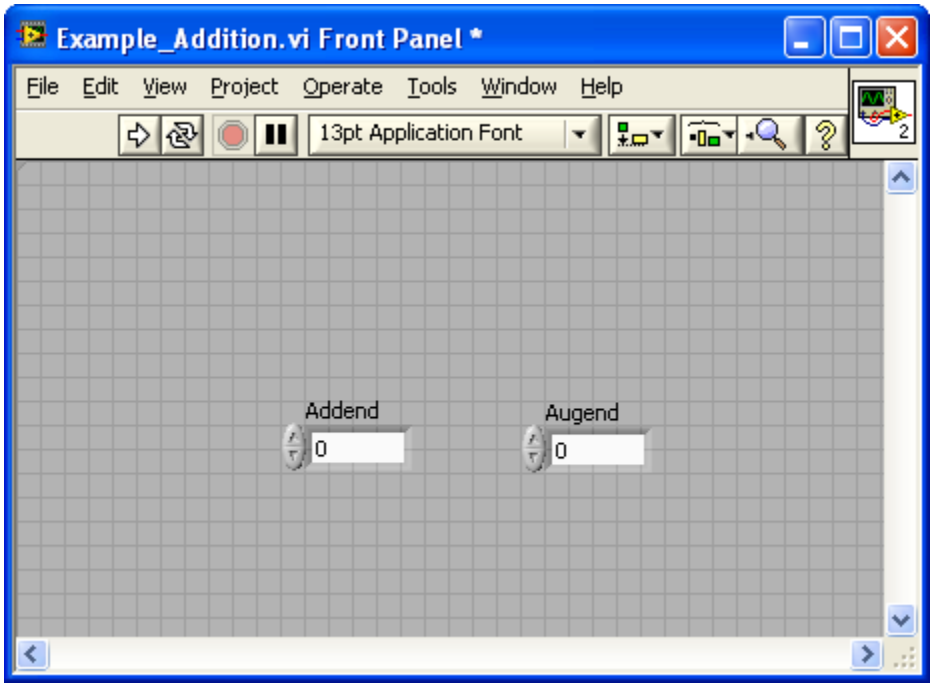


Figure 8: Two Numbers to Add

Now, from *Numeric Indicators* control group choose *Num Ind* (numeric indicator) and place it in the front panel to check result for the addition operation, as shown in *figure 9*

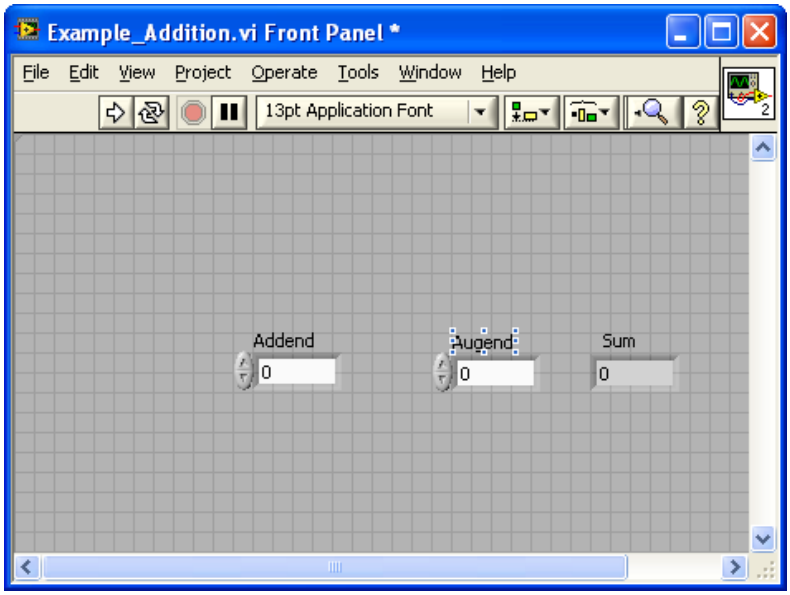


Figure 9: Numeric Controls with Number Indicator

Now add two labels, a '+' sign between *addend* and *augend*, and an '=' sign between *augend* and *sum*. This can be done by double clicking your mouse button where you want to place that text and typing it in the box that appears (if you are in *Automatic Tool Selection* mode).

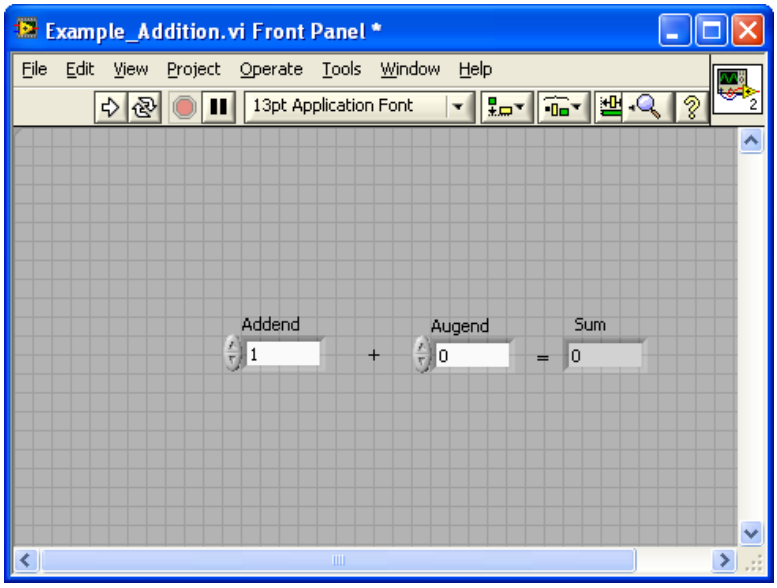


Figure 10: Adding text in Front Panel

Now go to your block diagram window and from *function palette*, choose an *ADD* function from *Mathematics* group and place it in window. Arrange your blocks and wire them as shown in *figure 11*. Run your circuit in the front panel and check results for different number additions.

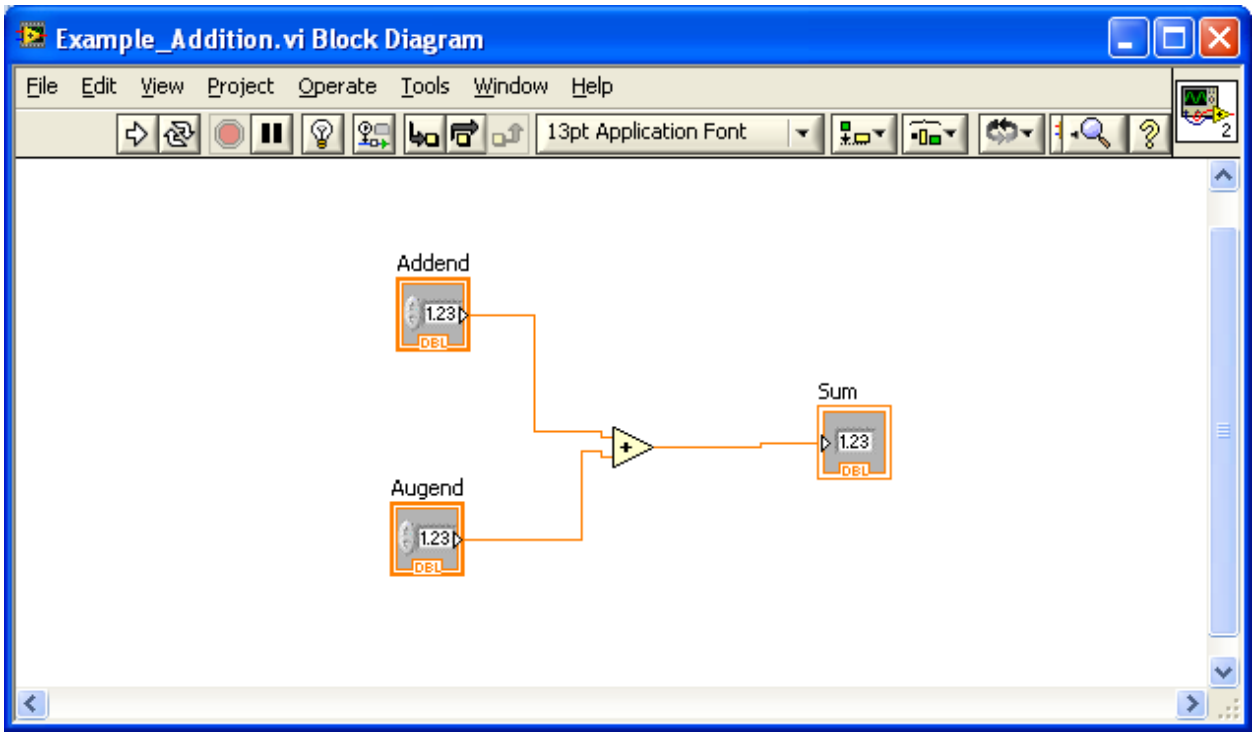


Figure 11: Addition Blocks

There is also a function block called *Compound Arithmetic* under *Numeric* group that can be used to perform multiple additions or multiple multiplications. You can stretch the block to have multiple inputs and you can choose the type of operation by clicking right on the block and choosing *change mode*. Use it when multiple additions or multiplications are required instead of using the block multiple times that can add or multiply only two inputs.

There is a nice option in LabVIEW that can show signal flow while you are simulating your circuit. This can be done by pressing *Highlight Execution* button from block diagram window (fifth button from left in the toolbar). You can also arrange your two windows right and left, or up and down from *Window* menu bar. Simulate your circuit with *Highlight Execution* button pressed to see how your values are flowing from your addend and augend to your sum block.

Function Evaluation

Let's evaluate the following function in LabVIEW,

$$f(x, y) = 4x + 3y \tag{1}$$

Place two numerical control blocks for *x* and *y*, and one number indicator block for *f(x,y)*, as you did in the previous design.

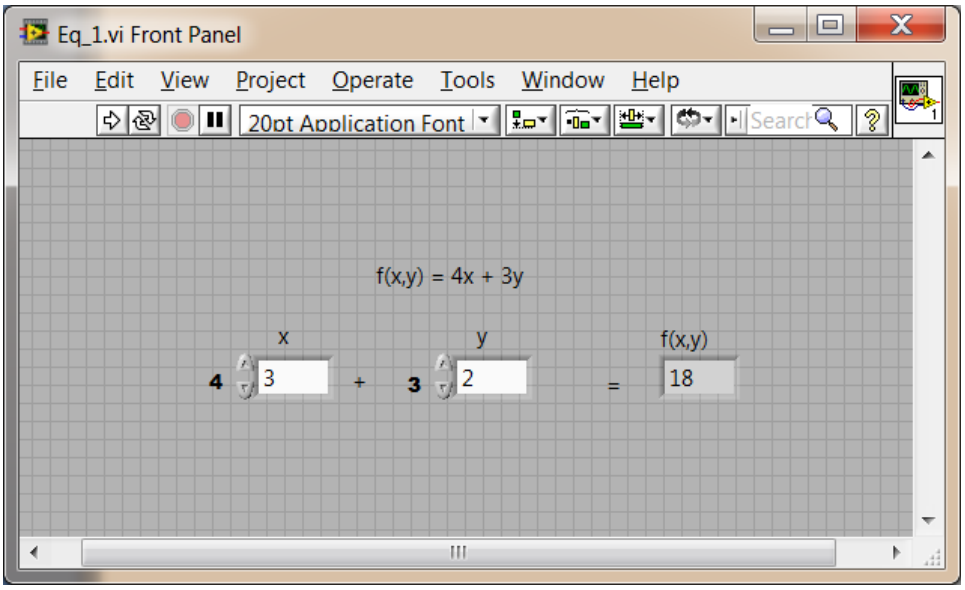


Figure 12: Control Blocks for Equation (1)

To present a constant in the block diagram, choose *Mathematics* under *function palette* and then proceed to *Numeric* → *DBL Numor Numeric* → *Numeric Constant*. Complete block for *equation 1* is shown in *figure 13*

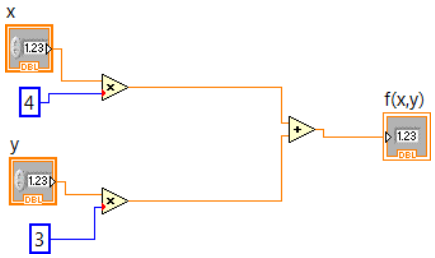


Figure 13: Block Diagram Corresponding to Front Panel from Figure 12

EXERCISE – 3

Implement the function,

$$f(x,y) = 4x^3 + 3y^2 + 7xy \tag{2}$$

Front panel controls are shown in *figure 13*

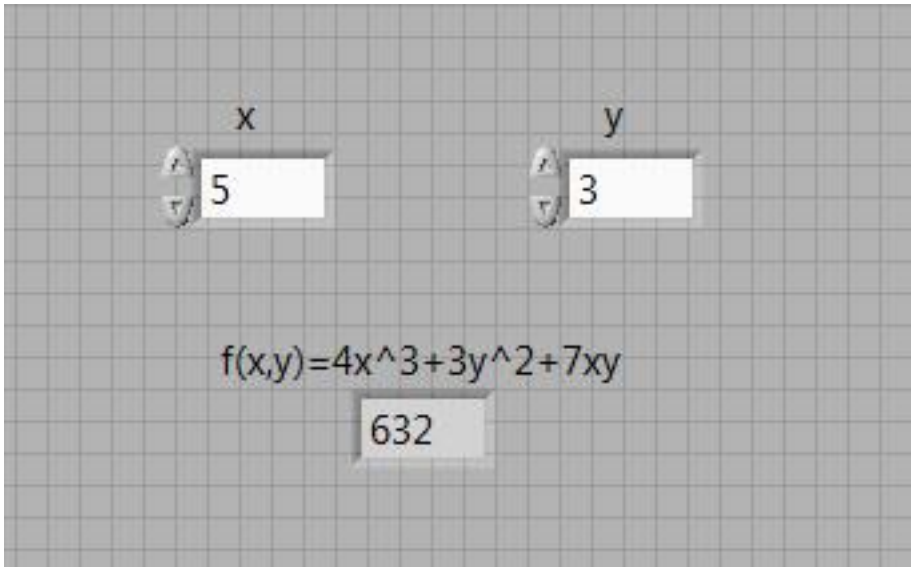


Figure 13: Controls Corresponding to Equation (2)

NOTE: x^2 block is in *Mathematics* → *Numeric* and x^yfunction block is in *Mathematics*→*Elementary*→*Exponential*

EXERCISE – 4

Enter temperature in Centigrade (from 0°C, freezing point of water, to 100°C, boiling point of water) with a *knob* and use *thermometer* to display equivalent temperature in Fahrenheit. Conversion formula is,

$$T_F = \frac{9}{5}T_C + 32 \tag{3}$$

NOTE: Use *right-click* → *scale* to set scale for both *knob* and *thermometer*

EXERCISE – 5

Design a control to find out voltage across resistor R_2 from the following circuit.

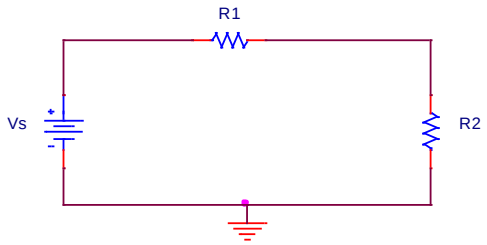


Figure 14: Simple Series Circuit for Exercise - 5

Use *voltage divider rule*,

$$V_{R_2} = \frac{R_2}{R_1 + R_2} Vs \tag{4}$$

Front panel controls are shown as follows:

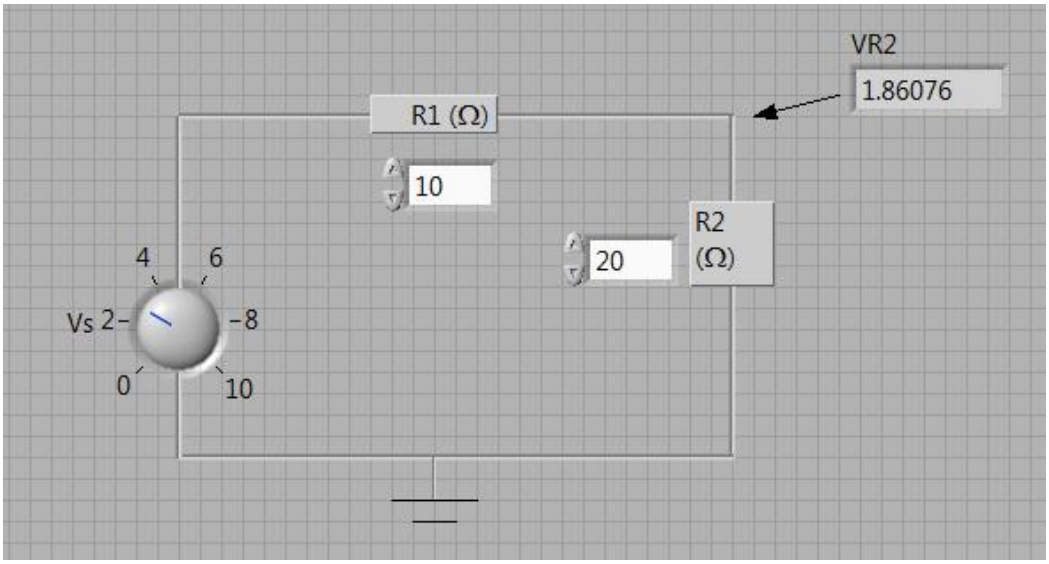


Figure 15: Front Panel Controls for Exercise – 5

NOTE: Use *Modern* → *Decoration* to draw lines and *R1* and *R2* boxes.

Descriptive Information& Block Diagram Labeling

This section discusses how to add a description to your program. Let's create a new VI that evaluates roots of a quadratic equation, $Ax^2+Bx+C=0$, using quadratic formula,

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{5}$$

Figure 16 shows the controls for the front panel

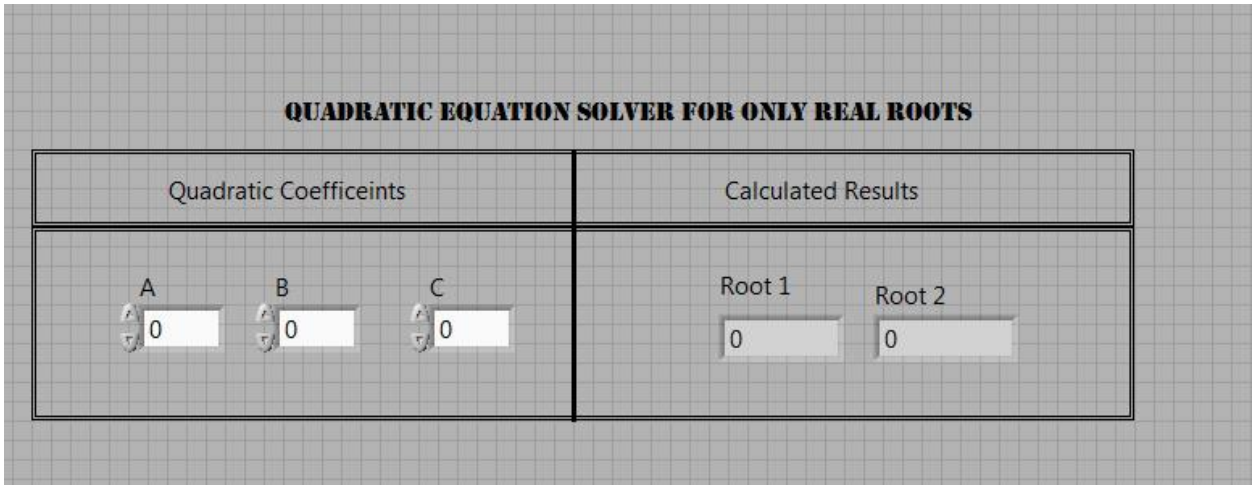


Figure 16: Quadratic Equation Solver for Real Roots

To add a description for this design, go to *File→VI Properties* and choose *Documentation* under *category*. You can also do the same by right clicking on the *LabVIEW* icon in the top right corner of your screen and choosing *VI Properties.* Give *VI* description for your program. One example is shown in *figure 17*

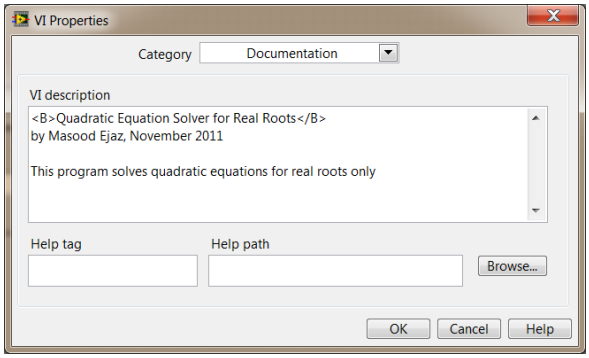


Figure 17: Example of File Description

You must have recognized <B>&</B> as used in *html* to print something in bold. Press OK to save the description. You can see *context help* about your program by moving your mouse over the *LabVIEW* icon on the top right corner of your screen, if context help is enable (To enable context help, go to *Help* → *context help* and then move your mouse on the icon)

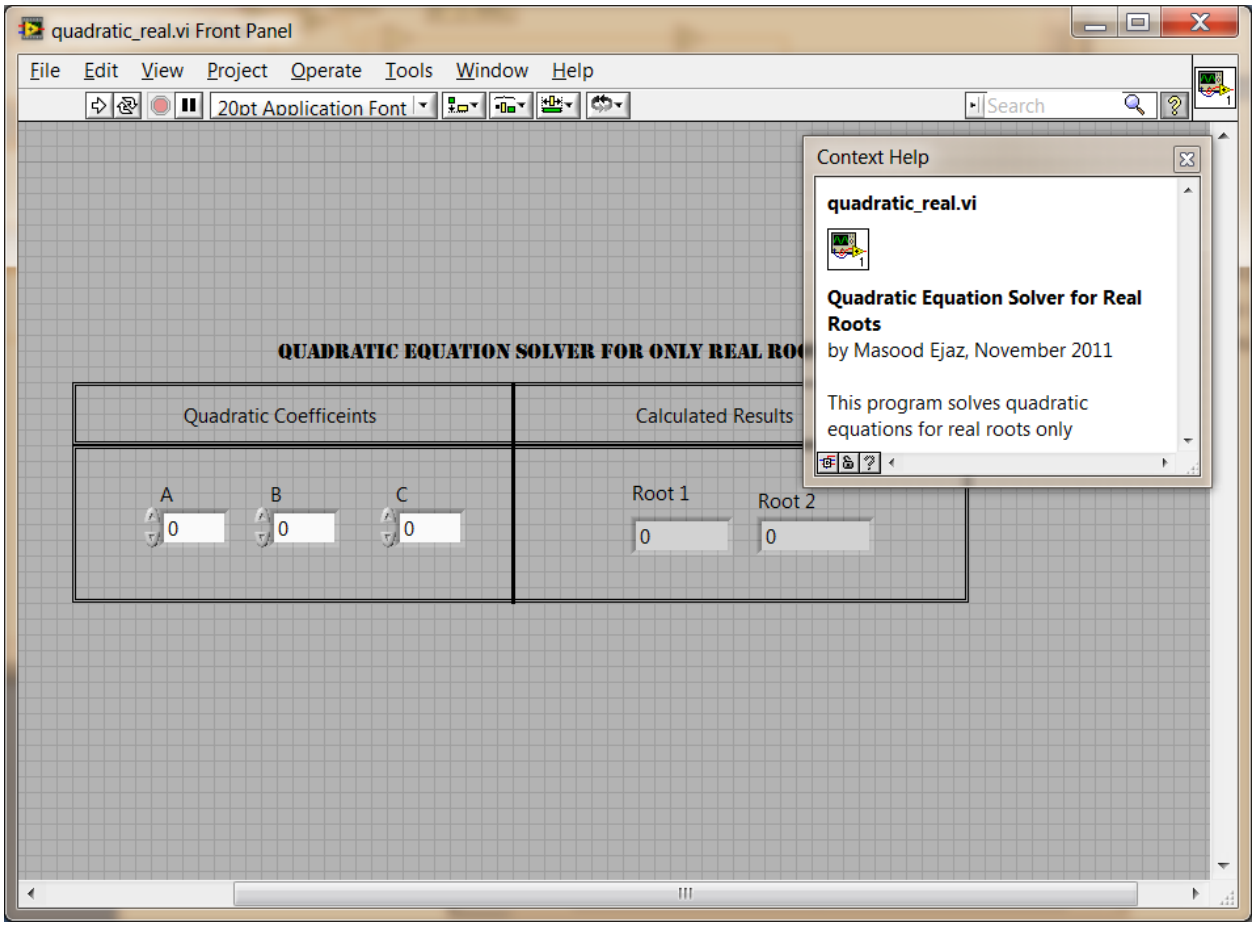


Figure 18: Context Help

Block diagram corresponding to the program is shown in *figure 19.* Observe the proper labeling that makes theblock diagram more understandable

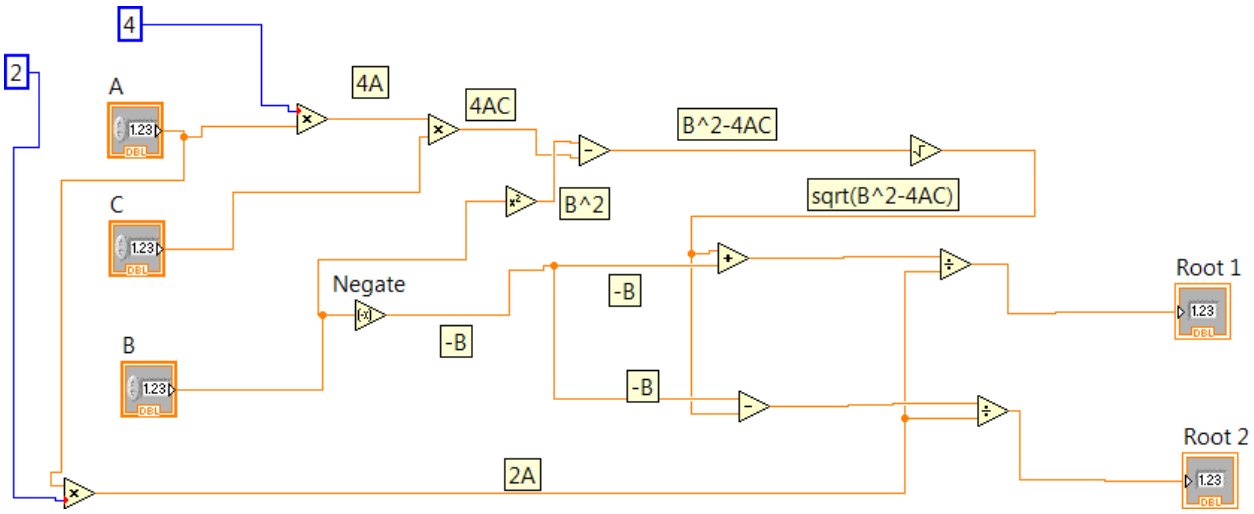


Figure 19: Block Diagram Corresponding to the Quadratic Equation Solver for Real Roots

To convert this program into a quadratic equation solver for both real and complex roots, all you have to do is to choose proper data type associated with the square root function block and the output *number indicator* blocks. This data type is *complex double* and you can choose it by right clicking the output number indicator blocks and going to *Representation* → *CDB*. Likewise, change the data type for the square root block from *Double* to *Complex Double (right-click* → *Properties* → *Output Configuration*). A proper quadratic equation solver that can solve for all the roots is shown in *figure 20*.

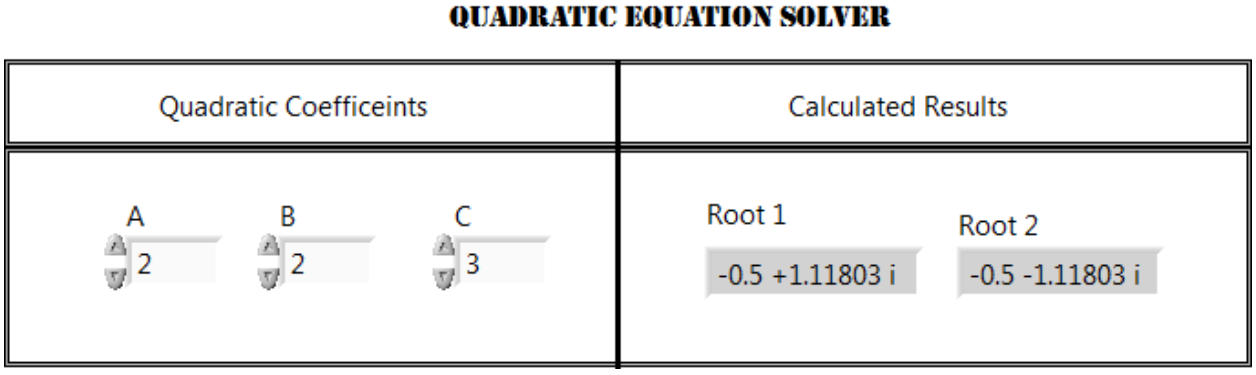


Figure 20: Quadratic Equation Solver for all Roots

Equation Evaluation using *Formula* Block

There are many function blocks given under *Mathematics* → *Script & Formulas*. Let's analyze one of them to see how we can evaluate equations and formulae without using basic mathematical function blocks.

Suppose we want to implement the equation $f(x,y) = 4x^3 + 3y^2 + 7xy$. In your front panel arrange numeric controls for *x* and *y* and number indicator for *f(x,y)* as shown earlier in *figure 13*. In block diagram window, pull *formula* block from *Mathematics* → *Script & Formulas*and arrange the blocks as shown in *figure 21*

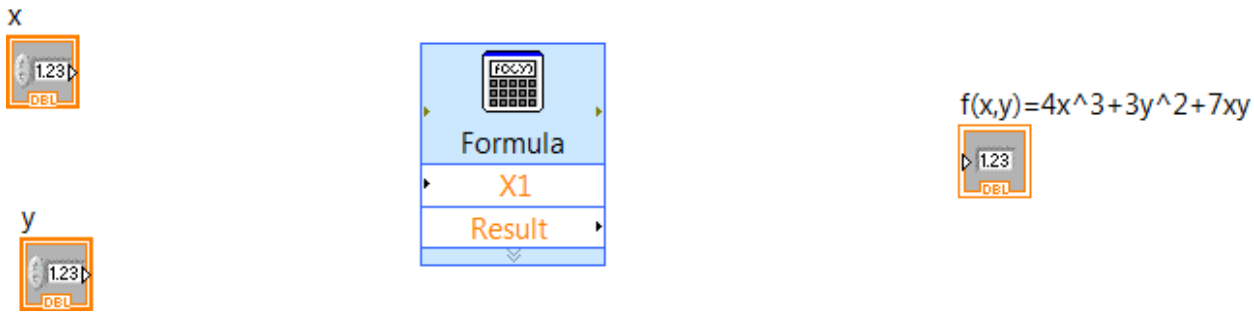


Figure 21: Arrangement of Blocks to Implement Equation (2) Using Formula Block

When you put the formula block, its properties window will open up that looks like a calculator. Type your formula in this calculator using the variables *X1* for *x* and *X2* for *y*. Make sure to use '**' button for power.Once you are done entering your formula, change the label of *X1* to *x* and *X2* to *y*. Press *OK* to exit from formula block properties. In the block diagram, your formula block will expand to show you two input connections, *x* and *y*. Connect inputs and outputs as shown in *figure 22*. Your control is ready to work!

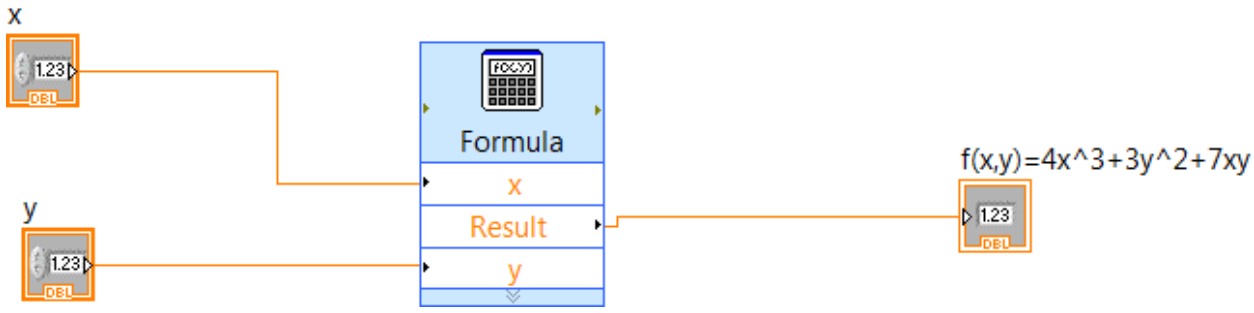


Figure 22: Circuit to Implement Equation 2

WARNING: Make sure not to use formula block to evaluate any expression or implement any type of function control unless you are asked to do so.

EXERCISE– 6

Design a binary to decimal number converter that converts a 3-bit binary number into its decimal equivalent. Take *C* as the most significant bit and *A* as the least significant bit. Front panel controls are shown in *figure 23*. When toggle switch is *ON* assume a *1* and when it is *OFF* assume a *0*, for binary number. Choose red switch color to show its *ON* state.

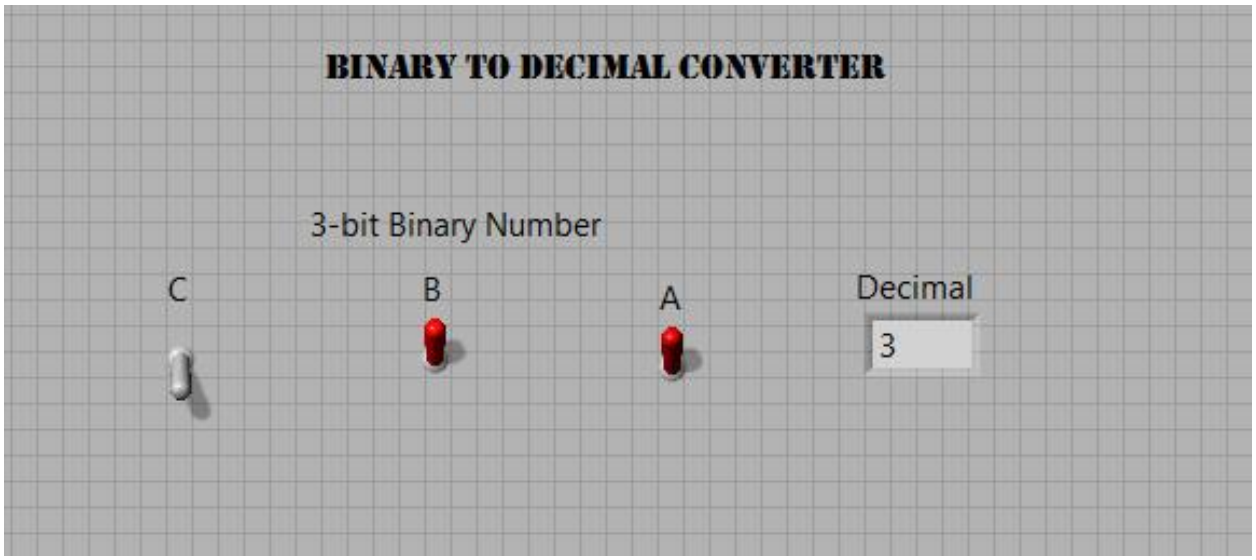


Figure 23: Binary-to-Decimal Converter

NOTES:

(i) You might need to convert *Boolean* logic of switches into equivalent *0* and *1* before you create conversion circuit. To do so, right-click on any switch in the block diagram and choose *Boolean palette* →*Boolean to (0,1)*. This is important to match input and output data types.

(ii) If you do not know how to convert a binary number into decimal, learn! It will be helpful in your future classes

Comparison Functions

Let's look at another important class of functions, *comparison functions*. Comparison functions compare different conditions at the input and generate an output based on the comparison result. These functions are located in *Express* → *Arithmetic & Comparison* → *Comparison*. You must be familiar with most of the functions in this group. Let's use a less familiar function *Select* to create a program.

Let's create a program that selects and calculates *natural log* or *base-10 log* of a number with a switch toggle. Front panel controls are shown in *figure 24*

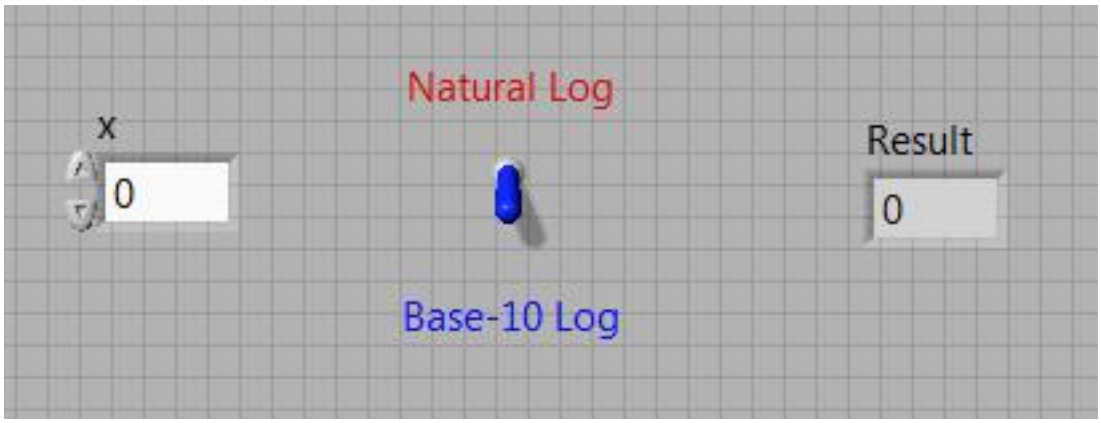


Figure 24: Toggle Switch to Calculate Corresponding Results for x

Pull *Select* block, *Log10* block, and *ln* block to the block diagram. Check *Select* block's *help* to understand it's operation. Wire them together as shown in *figure 25* and test your circuit.

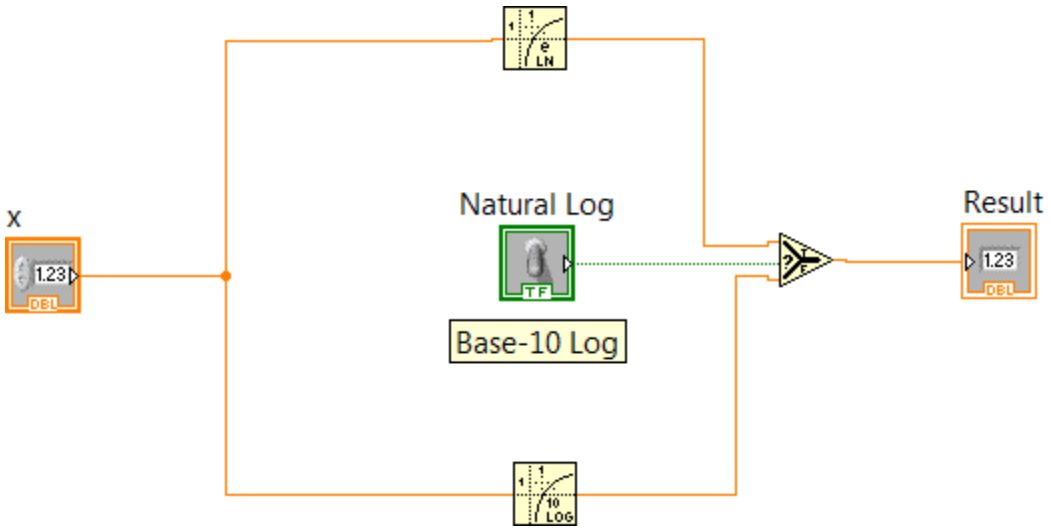


Figure 25: Block Diagram Corresponding to the Problem

EXERCISE – 7

Create a design to select one of the following functions with a toggle switch and calculate its output for different input values

$$f(x,y) = 2x + 3y$$
$$g(x,y) = 4x + 5y \quad (6)$$

Incorporating Other Comparison Functions with Select:

Let's create a program based on *Exercise -7* such that if input $x > y$, output will be calculated according to *f(x,y)*, otherwise, it will be calculated according to *g(x,y)*. We are going to use two comparison function blocks; *greater than* and *less or equal to* along with *select* block. Make sure to check their help to see the data types associated to their inputs and output.

Open the front panel for *Exercise – 7*, remove the switch and rearrange the controls as shown in *figure 26*. Add two LEDs, one to show when result will be calculated using *f* and other when it will be calculated using *g*.

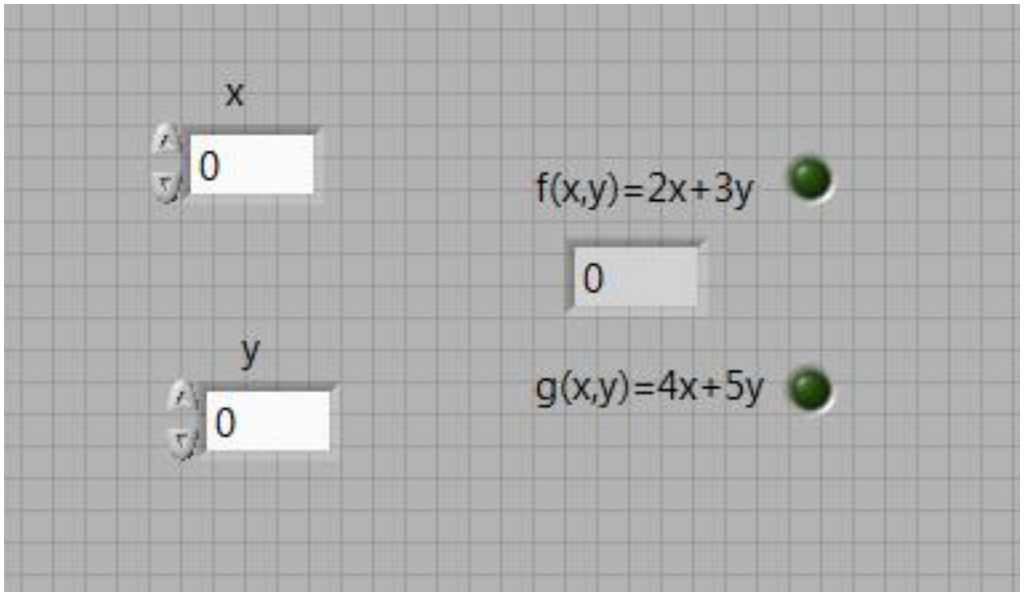


Figure 26: Calculation of Two Different Functions Based on the Input Values

Block diagram of the control is shown in *figure 27*. Note how the output of *greater than* block is connected to both of the LEDs directly. Practically, you will need an inverter circuit (*NOT* gate) connected to the LED of *g(x,y)* function but just for demonstration purpose, you do not need to connect any gate between the output of comparison block and LEDs. How you are going to make sure that only one LED will light up when the corresponding function will be calculated, as shown in *figure 28*?

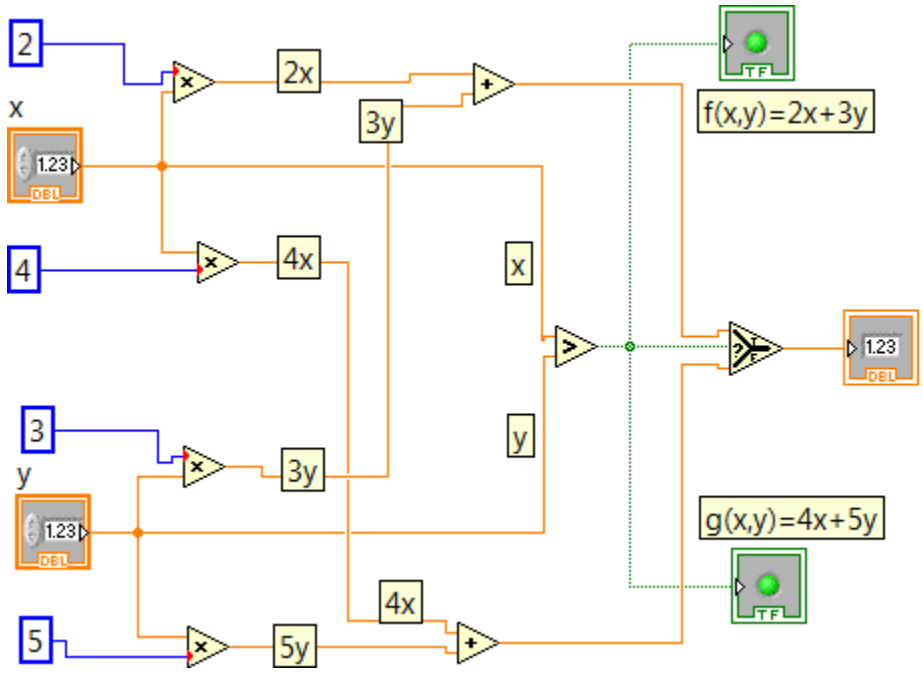


Figure 27: Block Diagram for Figure 26 Circuit Control

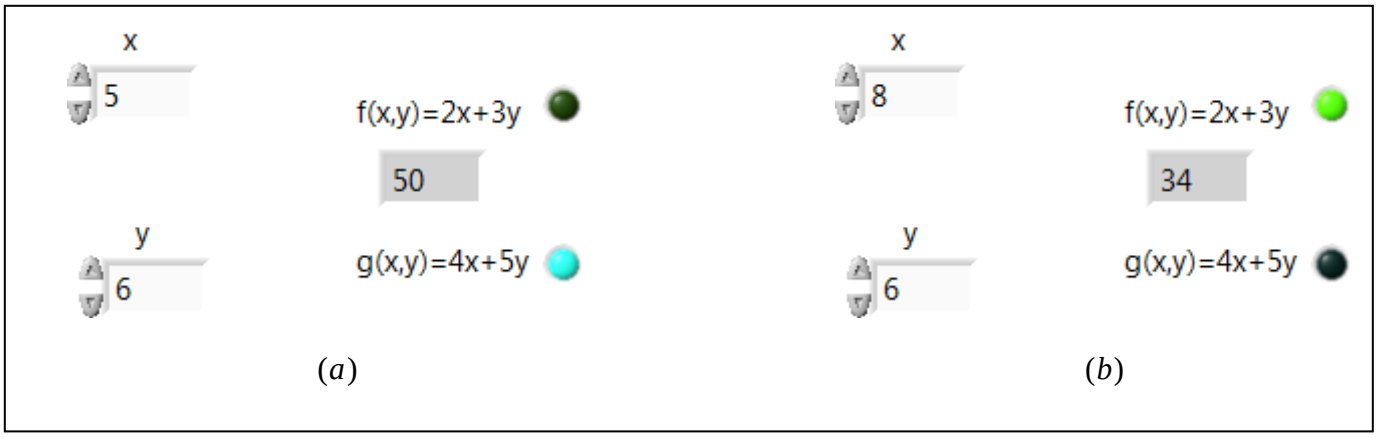


Figure 28: Output of Two Functions, Selected One at a Time, Base on the Input Values (a) Output is Calculated for g(x,y), since x < y (b) Output is Calculated for f(x,y), since x > y

Random Number Generation:

There is a random number generator in *function palette* under *Mathematics* → *Numeric* → *Random Number (0-1).* This random number generator generates a floating point number randomly, using a *uniform distribution function,* between 0 and 1. If you want to generate any number between a specific range, say 0 to 10, all you have to do is to multiply the output of random number generator by the upper limit of your range. If you want output to be only random integers, instead of floating point numbers, round them off to the closest integer using the appropriate function block

EXERCISE – 8

Create a program that randomly generates an integer between 0 and 10. Front panel will contain only one *number indicator*

Introduction to WHILE Loops

Suppose we want to create a program that a user has to guess a randomly generated number between 0 and 10 from the program that you just created. This calls for user to keep entering numbers until his number matches with the one randomly generated. This situation can be handled with a *WHILE* loop. WHILE loops perform a set of functions as long as some condition is *true* or *false*. Since WHILE loops are related to comparing results between different conditions, hence this is a natural place to introduce them.

WHILE loop function block is present under *Programming* → *Structure*. Pull the function block on the block diagram. It looks like a box that can be resized. Anything that you enclose inside this box will be repeated until condition of WHILE loop becomes *true* or *false,* however you set it. Make sure to check help for WHILE loop to fully understand its function.

Let's create a program to let user guess a randomly generated number between 1 and 10. Front panel will contain a *numeric control* (disable *increment/decrement* control so that user can only enter the number by typing it) and an *LED* as shown in *figure 29*

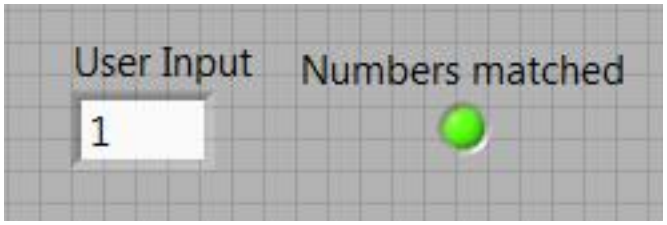


Figure 29: Front Panel for Number Guessing Program

We will enclose *user input* and comparison block inside the WHILE loop and randomly generated number control outside the WHILE loop, since random number has to be generated only once. LED will also be outside the WHILE loop, connected to the output of comparison block. It will turn *ON* when WHILE loop condition will be satisfied, i.e. when input number will be matched with the random number. Any indicator outside the WHILE block will only demonstrate its status once the WHILE condition is satisfied. Block diagram of the program is shown in *figure 30*

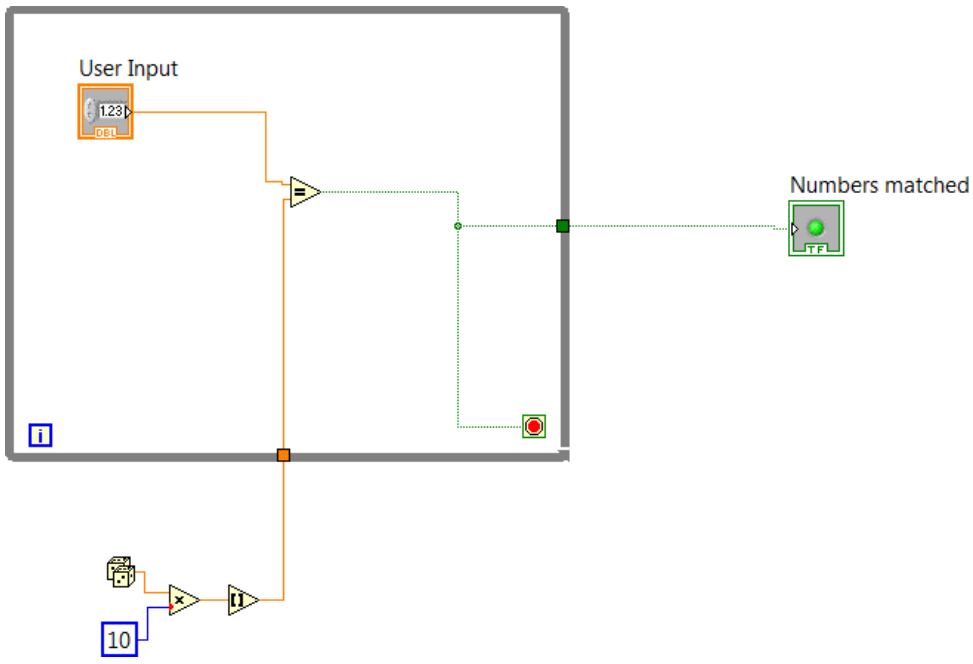


Figure 30: Block Diagram for Number Guessing Program

Condition for the WHILE loop that is used is *Stop if True*. You can change the condition to *Continue if True* (right-click WHILE block) and change *equal to* block to *not equal to*, to get the same result.

Boolean Functions:

Boolean or Logic functions are comprised of basic logic gates (*NOT, AND, OR)* and advanced logic gates (*NOR, NAND, XOR, XNOR).* Boolean function group also has some other function blocks that can be used as required. Just to refresh your memory, functions of different logic gates are as follows,

- *NOT:* Output is opposite of the input
- *AND:* Output is high only if all the inputs are high
- *OR:* Output is low only if all the inputs are low
- *NAND:* Opposite of *AND.* Output is low only if all the inputs are high
- *NOR:* Opposite of *OR.* Output is high only if all the inputs are low
- *XOR:* Output is high if both of the inputs are opposite
- *XNOR:* Output is high if both of the inputs are same

Let's start with an easy example. Choose two dials *A* and *B*, each with range from zero to ten. Choose two LEDs, one for each dial. If A > B, turn LED for dial *A* ON, else turn LED for dial *B* ON. This can be made with a simple *NOT* gate. Front panel for the program is shown in *figure 31* and block diagram is shown in *figure 32*.

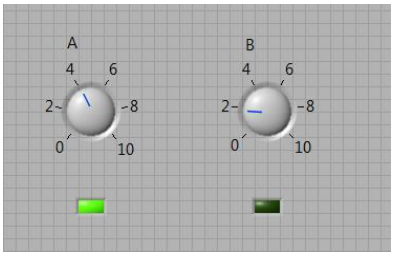


Figure 31: Two Dials Controlling Two LEDs

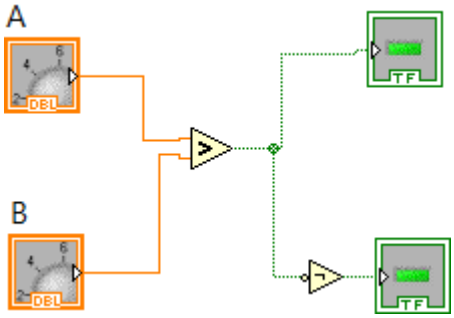


Figure 32: Block Diagram for Two-Dial Circuit from Figure 31

EXERCISE – 9

Create a program that can generate random integers from 3 to 8. Your front panel will have only one numeric indicator to show random integer.

Hint: One way to do it is to use WHILE loop as part of your program

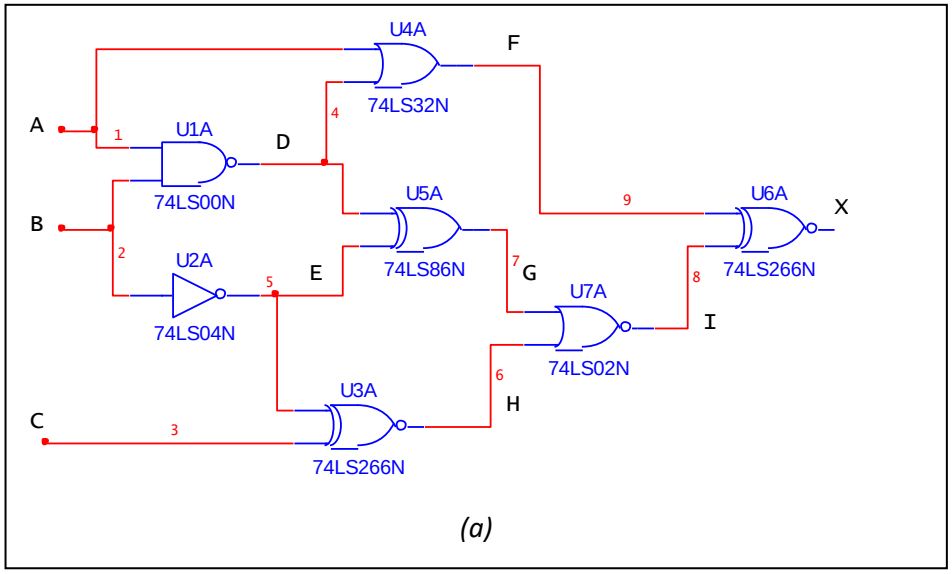
EXERCISE – 10

Choose three dials *A*, *B*, and *C*, each with range from zero to ten. Also choose three *LEDs* (different colors), one for each dial. Design a control circuit that shows you which dial has the highest value by turning ON the corresponding LED

Hint: One way to design this control is to use *AND* gates as part of the design

EXERCISE – 11

Create the logic circuit shown in *figure 33* and fill out the following truth table. Use three switches for the inputs and an LED for the output. Symbols for different gates are given in *figure 34*



(a)

A	B	C	X
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)

Figure 33: (a) Logic Circuit (b) Truth Table to Fill

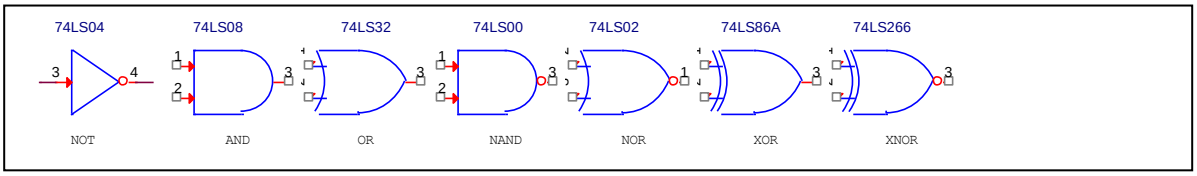


Figure 34: Symbols for Different Gates

Matrices & Arrays

In LabVIEW, matrices represent data arrangement in *2D* while arrays represent data arrangement in any dimension. In arrays you can store data with only one data type (double floating, Boolean, integer etc.). If you want to store data with different data types (for example, a mix of floating, image, integer etc.), *clusters* are used instead of arrays.

To define a matrix, go to *Array, Matrix, and Cluster* control group and choose *RealMatrix*. Let's add two matrices as follows,

$$\begin{bmatrix} 2 & 3 & 9 \\ 1 & 6 & 4 \\ 8 & 5 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 3 & 8 \\ 2 & 5 & 2 \\ 1 & 3 & 2 \end{bmatrix}$$

Pull two matrix controls on the front panel, name them*A* and *B*. First index of matrix control represents row number and second one represents column number. Note that both row and column numbers start at *0*. You can remove vertical and horizontal scroll bars and index control by going into the properties of matrix block. Also, you can stretch matrix block to accommodate visible number of rows and columns. Front panel is shown in *figure 35*. Make sure to change *A+B* matrix from *control* to *indicator* (*right-click → change to indicator*).

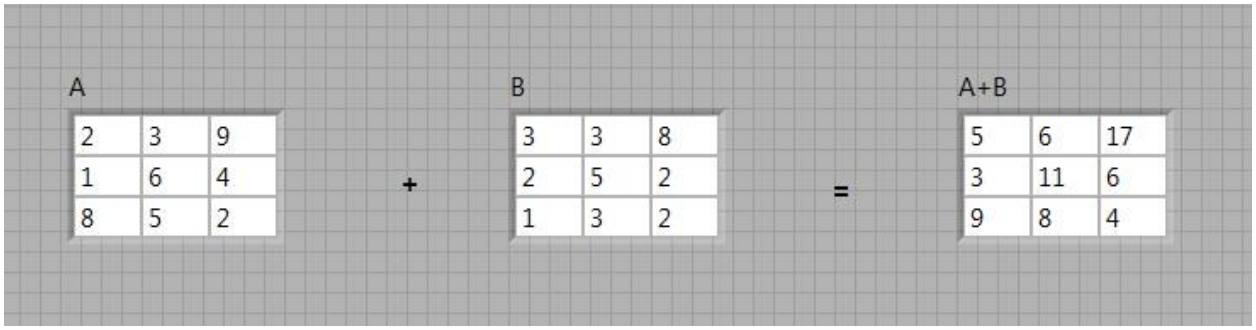


Figure 35: Matrix Addition

Explore different functions of matrix block by highlighting and right-clicking it.

Now let's define an array. Pull *array* block from the corresponding control group. It will look like an empty box. Insert a *numeric indicator* in the box and stretch it to control number of rows with one column. To increase number of columns, right-click the block and click on *Add Dimension*. Column dimension will appear. Stretch the array block to adjust number of columns.

When you use matrix block to do matrix multiplication, LabVIEW carries out *cross product*. Hence, number of columns of first matrix should be equal to number of rows of the second matrix. With arrays, you can either do element-by-element multiplication (array multiplication) or cross multiplication by using the corresponding block from *Mathematics* → *Linear Algebra* → *A* × *B*.

EXERCISE – 12

In circuit analysis, when *KVL* (Kirchhoff's Voltage Law) or *KCL (*Kirchhoff's Current Law) equations are set up to solve for either loop currents or node voltages, it results in a set of simultaneous linear equations. Solve the following set of simultaneous linear equations using matrix manipulation as given below.

$$2x + 5y + 6z = 2$$
$$3x + 6y + 2z = 1$$
$$8x - 5y + z = 4$$

This set of linear equations can be represented in terms of matrices as follows,

$$\begin{bmatrix} 2 & 5 & 6 \\ 3 & 6 & 2 \\ 8 & -5 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}$$

Values of unknown variables can be found out as follows,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 & 5 & 6 \\ 3 & 6 & 2 \\ 8 & -5 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}$$

where $\mathbf{A}^{-1}$ represents matrix inverse. There is an *inverse matrix* function block in LabVIEW (find it!). To keep your values intact in any matrix, right click on the matrix, go to *data operations* and select *make current values default*. If you do not make them default then they will be lost once you close your file.

There is also a linear equations solution function block under *Mathematics → Linear Algebra → Solve Linear Equations*. Use this block and work on *Exercise – 12* again.

Another important function related to matrices and arrays is to extract different columns and rows and create new matrices or arrays out of them. The function that is used to do this job is *Array Subset* (find it!). Function block is shown in *figure 36* with description of its inputs and outputs. As with any array control block, by default *Array Subset* block is one-dimensional (only rows). You can always add more dimensions by selecting *Add Dimension* through right-click or as soon as you will connect a matrix to the input of function block, it will automatically extend its input dimension. In *figure 36*, both rows and columns dimensions are shown (blue inputs)

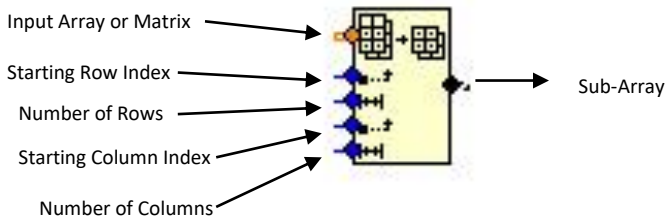


Figure 36: Array Subset Block

Create a 3-by-3 matrix or array and use *array subset* block to extract a two-by-two matrix with rows 2 & 3 and columns 2 & 3 as shown below. Remember the first column and row indices in LabVIEW are zero.

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 7 \\ 9 & 10 \end{bmatrix}$$

Origianl matrix *Extracted matrix*

Figure 37: Matrix and Sub-Matrix

There is also a very useful function to find out maximum and minimum value (s) of a matrix or array and indices of the first maximum values. This function is *Programming → Array → Max & Min*

Two-Dimensional Graphs:

Graphs utilize arrays to hold the values of the coordinates. Let's start with a simple line plot of equation $y = 2x + 1$ for $x = 1$ to 10,

$x = 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10$

$y = 3 \quad 5 \quad 7 \quad 9 \quad 11 \quad 13 \quad 15 \quad 17 \quad 19 \quad 21$

Create two arrays, one for *x* and one for *y*. There are two types of *X-Y* plot functions that you can use; *XY Graphs* and *Express XY Graphs*. Let's start with *ExpressXY Graphs*(*Modern → Graphs → Express XY Graphs* or *Express → Graph Indicators → XY Graph)*. Click on the *x* and *y* axes range to set initial and final values for both the axes. You can also click on the *x* and *y* labels to change them as well as the title of the plot. Front panel is shown in *figure 38* and block diagram is shown in *figure 39.*

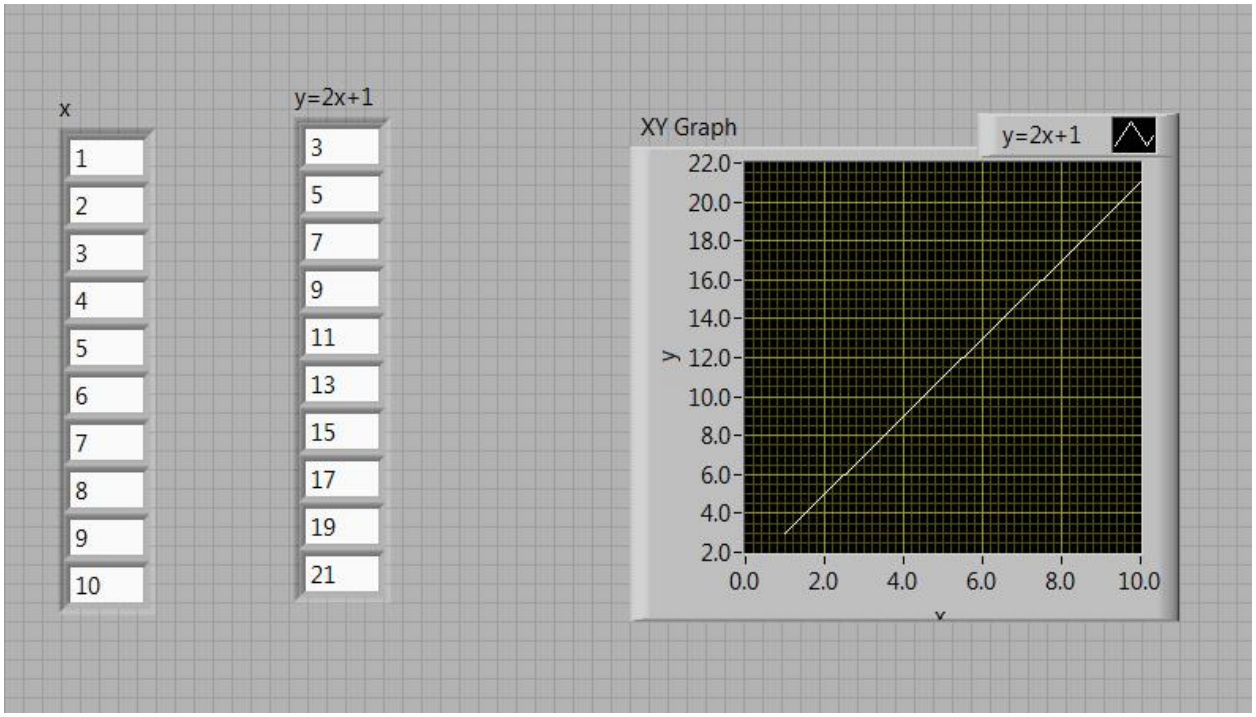


Figure 38: Simple Line Function Graph

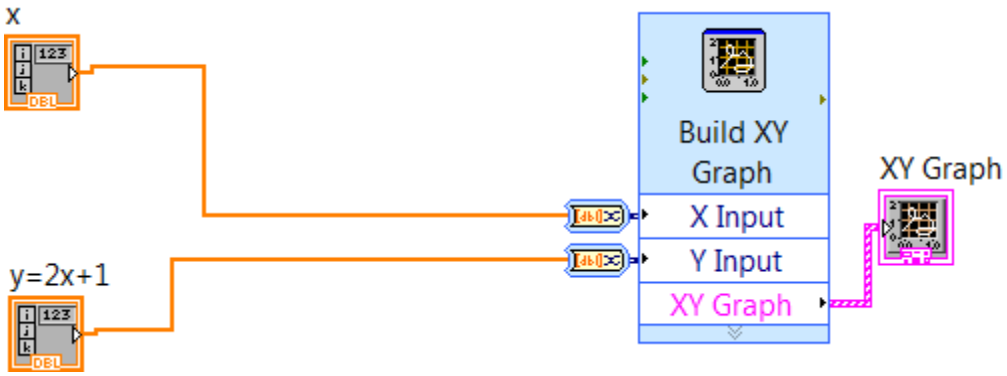


Figure 39: Block Connection for Line Graph from Figure 38

As it can be seen, express graphs have a *Build XY Graph* function added to them that lets you put two input arrays as *x* and *y* and generates an *XY Graph* between them.

If you use *XY Graph (Modern → Graphs → XY Graphs)*instead of *Express*, you have to create an *array bundle (Programming → Cluster, Class, and Variant → Bundle)* for the input arrays *x* and *y* such that output is a single array going into graph function. Recreate the line function graph using *XY Graphs* and *Bundle* function. Block diagram is shown in *figure 40*

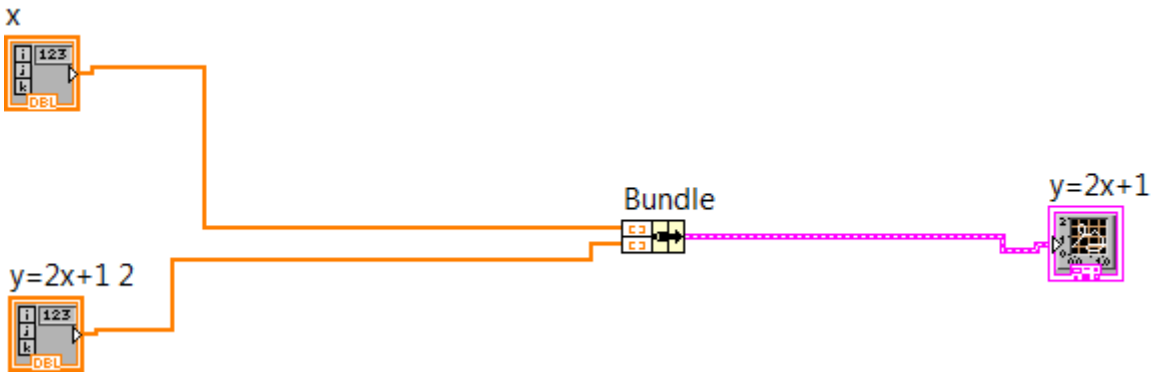


Figure 40: Graph of Straight Line Function using XY Graphs and Bundle Function

For Loops:

For loops are used to generate values for a function for a range of its input variable(s). For loop function block is located under *Programming* → *Structures* and shown in *figure 41*. Operation of this block is very simple; loop is going to run for *N* times, with *i* being the current iteration value from 0 to *N-1*. Arrays are generally used to hold the values and results generated from the loop.

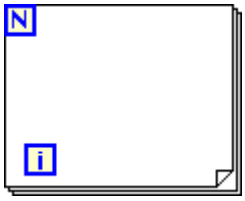


Figure 41: For Loop Function Block

Let's create a very simple program; build an array with values from 1 to 100. This can be done easily using a *for loop* as shown in *figure 42*.

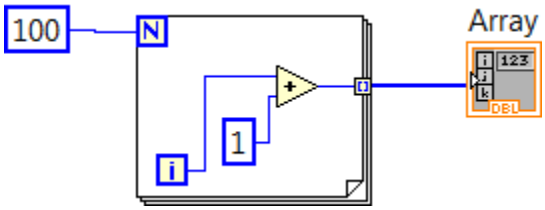


Figure 42: Using For Loop to Create an Array from 1 to 100

EXERCISE – 13

Use a*for loop* to generate a multiplication table from 1 to 10 for an integer *n*. Front panel is shown in *figure 43*.

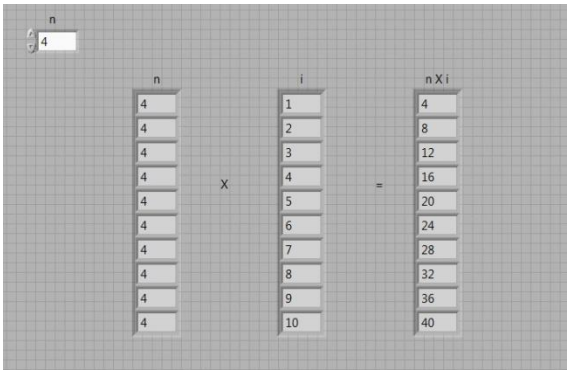


Figure 43: Front Panel for Multiplication table

Graphs Using For Loops:

One of the most important functions of *for loops* is to generate arrays to plot graphs. Let's create a plot for a quadratic function, $y = 2x + 3x^2 + 1$ for $x = 0$ *to* 20. Loop is going to run for *21* times ($N = 21$). Block diagram of the program is shown in *figure 44*and graph is shown in *figure 45*.

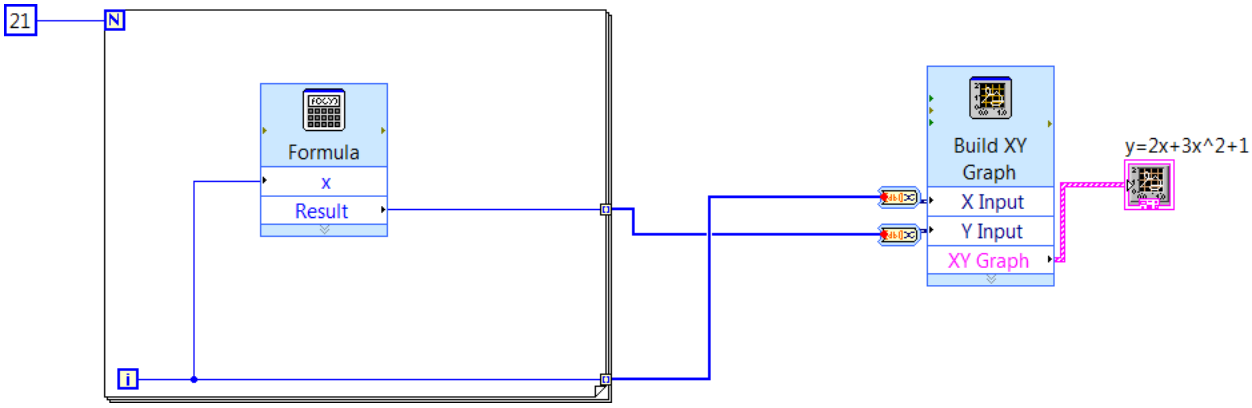


Figure 44: Block Diagram forthe Quadratic Function Plot

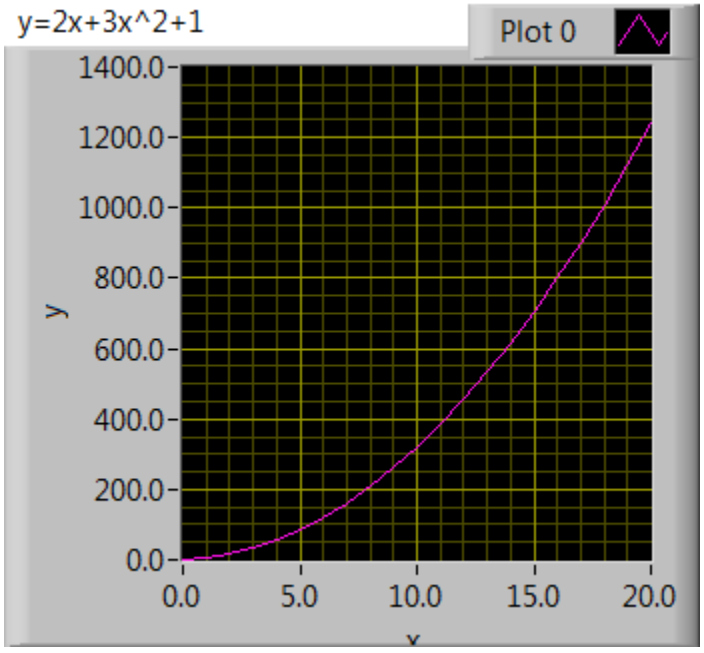


Figure 45: Plot of Quadratic Function $y = 2x + 3x^2 + 1$

EXERCISE – 14

Create a graph for $y = sin(x)$ for $x = 0^o$ *to* 360^o. Make sure your *x-axis* should be in degrees.

Multiple Graphs

In this section we will look at how to graph multiple functions on the same plot. Let's start with express XY graphs and plot two functions, $y_1 = 2sin(t)$ and $y_2 = cos(t)$, on the same plot. Let's plot both the functions from 0 to 4π with 100 points distributed evenly in the range. Front panel set-up will be the same as shown in *figure 46*. In block diagram you have to run the FOR loop 100 times such that it produces input to the trigonometric functions that starts at time $t = 0$ and ends at $t = 4\pi$. This can easily be done by dividing 4π by 99 (maximum value of i) and multiplying it by i as shown in *figure 46*. This value will go to the inputs of both *sin* and *cos* blocks.

To draw multiple graphs on the same plot, we will use a function block called *Build Array*. This is located under *Programming* → *Array*. You will need two of these blocks, one to build array for *x* (time) and the other to build array for *y1* and *y2*. Output of respective blocks will go to the *x* and *y* inputs of *Build XY Graph* block. Complete block diagram is shown in *figure 46* and front panel shot is shown in *figure 47*.

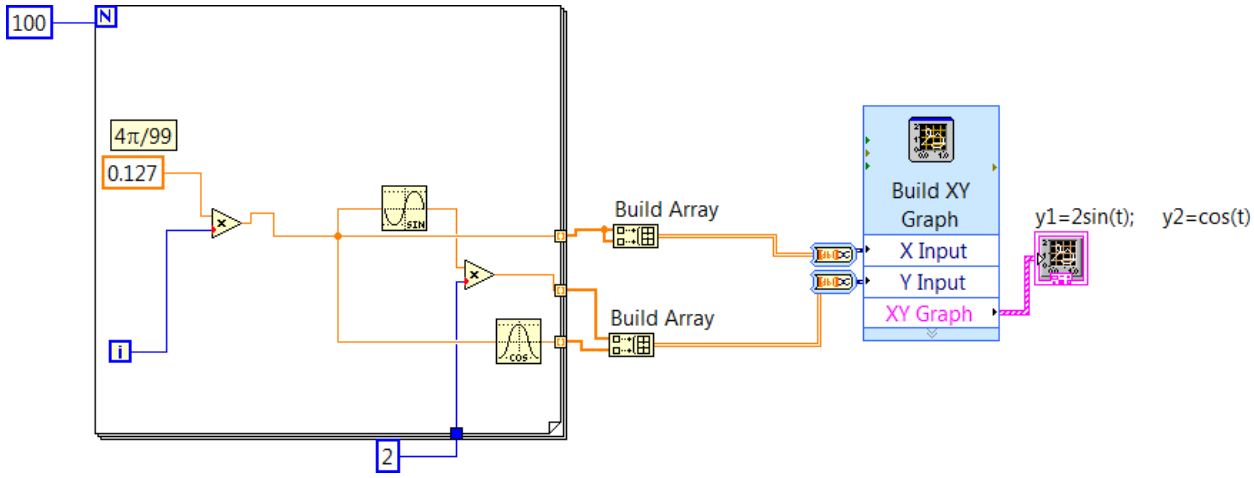


Figure 46: Block Diagram to Graph Two Functions on the Same Plot

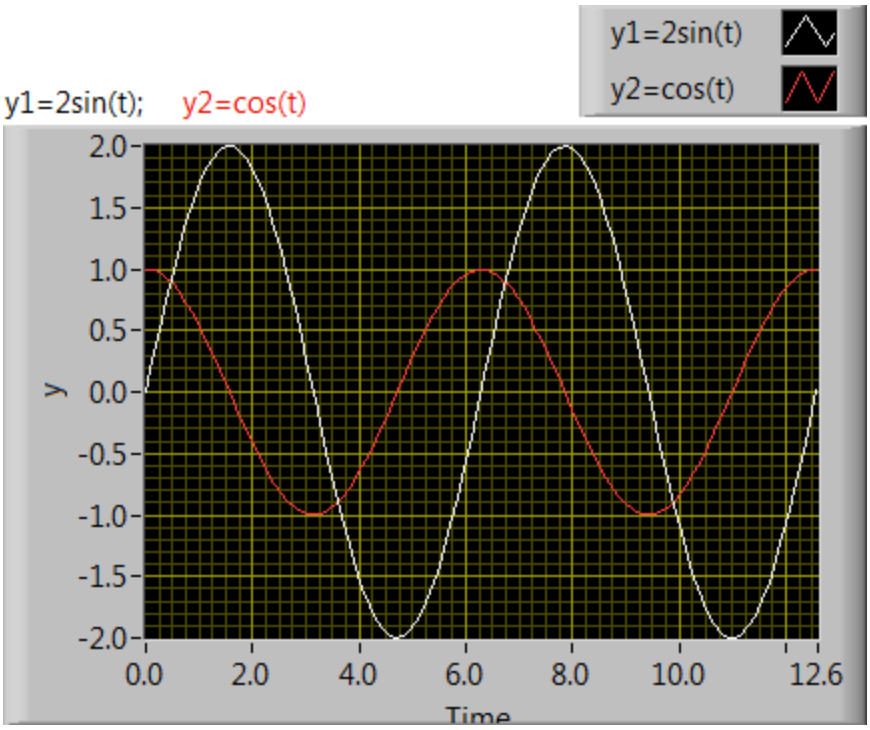


Figure 47: Multiple Graphs on the Same Plot

EXERCISE – 15

Plot the same functions y_1 and y_2 using *XY Graph* (not *Express XY Graph*)

Hint: You will have to use *bundle* block to make pairs of **(x, y_1)** and **(x, y_2)**first**,**followed by *build array* block

Cursors

You can add cursors to check the (*x,y*) coordinates of each graph. Turn the cursor on by right clicking the graph and selecting *Visible Items* → *Cursor Legend*. By default there is only one cursor. To add more cursors, right click on the graph again, go to *Properties* → *Cursors* → *Add.* You can change color of each cursor lines to differentiate from one another. An example is shown in *figure 48* that shows coordinate values of two graphs at different points

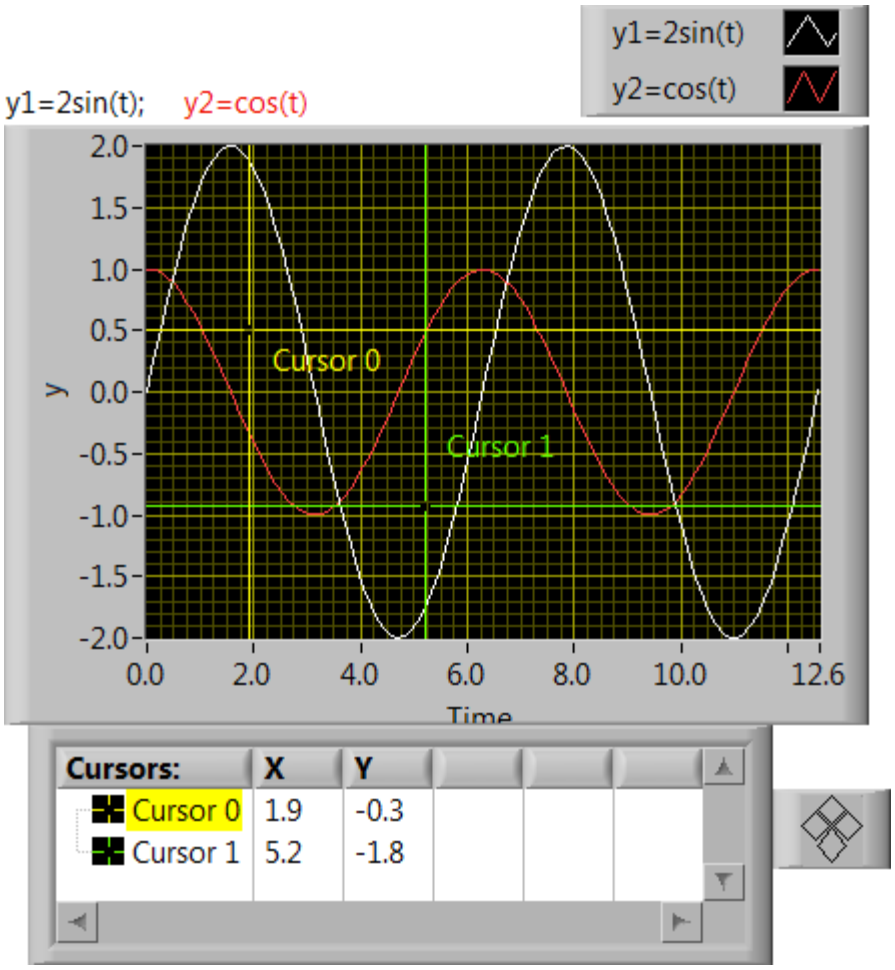


Figure 48: An Example of using Cursors

Waveform Graphs

Waveform graphs are very similar to *XY Graphs* except *XY Graphs* are a little more powerful. The waveform graph displays one or more plots of evenly sampled measurements. The waveform graph plots only single-valued functions, as in $y = f(x)$, with points evenly distributed along the x-axis, such as acquired time-varying waveforms.On the other hand, the XY graph is a general-purpose Cartesian graphing object that plots multivalued functions, such as circular shapes or waveforms with a varying time base. The XY graph displays any set of points, evenly sampled or not.

Since we are already familiar with *XY Graphs*, we are not going to talk about *Waveform Graphs* and it will be left as students' exercise.

EXERCISE – 16

Graph a function $y = Ax^2+Bx+C$, where A, B, and C are three dials, each with range from -5 to 10. Use *Waveform Graph* to graph y for x = 0 to 10.

Waveform Chart

Unlike graphs that wait for the output array to have all of the values and then create a plot, waveform chart receives individual data points and continuously updates the presentation of the data. A waveform chart is typically used during data acquisition to monitor the data as they are being collected. Let's modify *exercise 16* and use a *waveform chart* in addition to a *waveform graph* as shown in *figure 49*. Both of them are connected to the same function *y*.

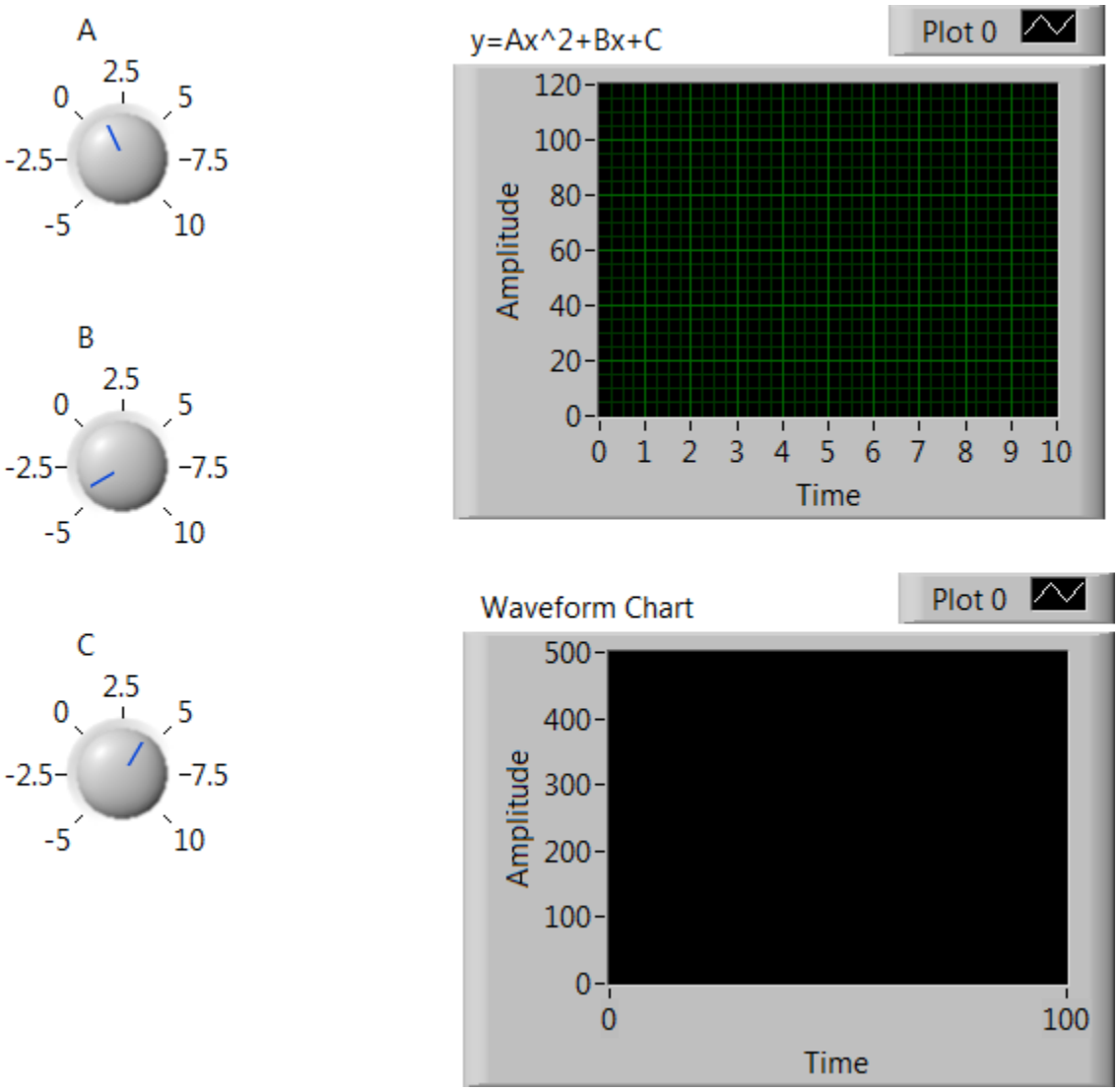


Figure 49: Demonstration of Waveform Chart

After running the program two times and keeping the same values of *A*, *B*, and *C*, you will see the following output

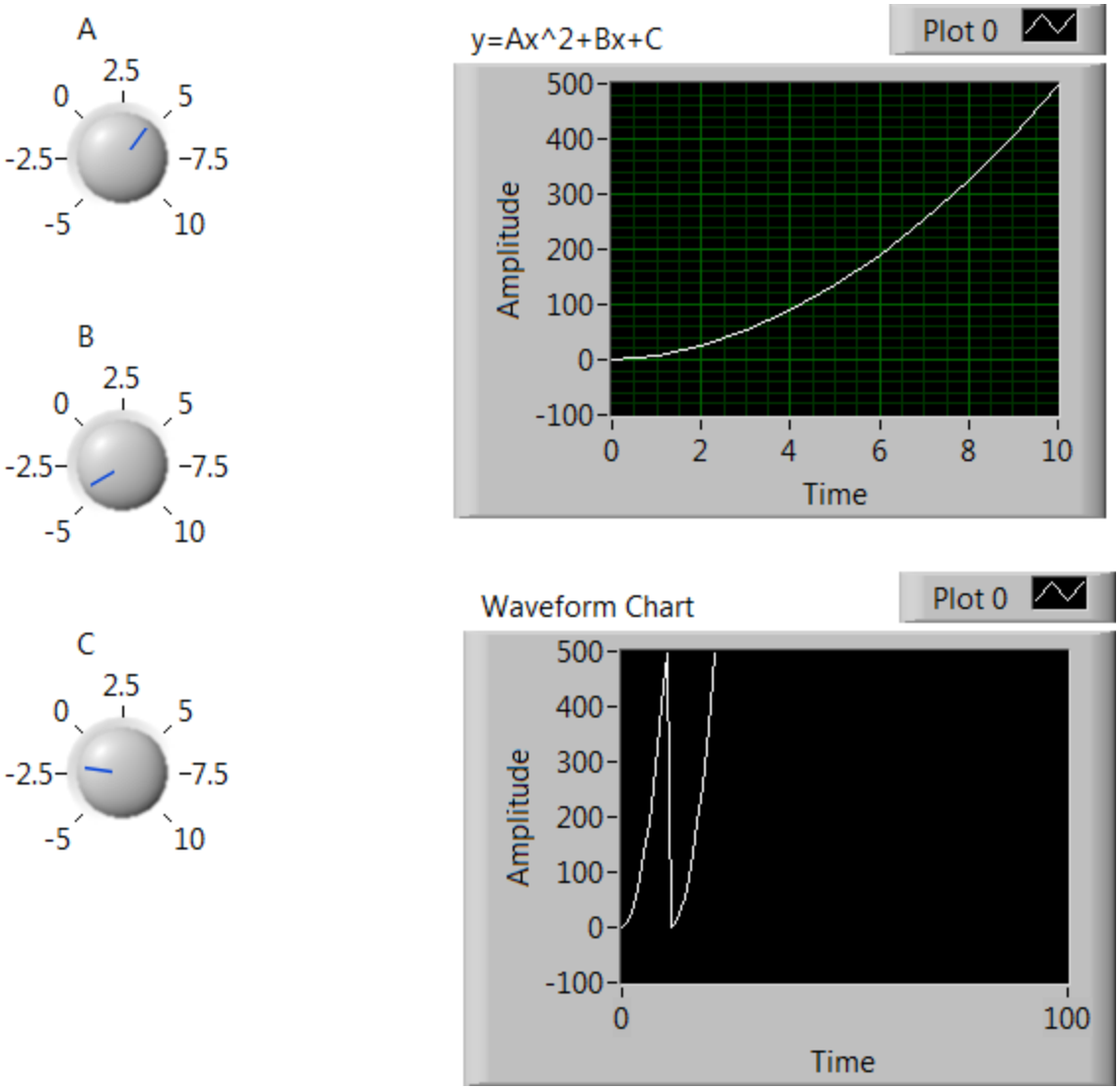


Figure 49: Difference in the Outputs of Waveform Graph and Waveform Chart

As it can be seen that *waveform graph* just plots the graph in the same range for each run of the program, whereas, *waveform chart* updates the range each time it runs. Note that function is still being calculated for the same values of *x* (0 to 10), only presentation is changed. *Waveform chart* keeps appending the range each time it is executed; hence, it is mostly used for data acquisition problems where continuous plot of incoming data is required.