BE/EE/MedE 189a: Design and construction of biodevices

Justin Bois

Caltech

1 LabVIEW basics

In this class, we will use a National Instruments ELVIS II breadboard to build out devices. We will connect these breadboards to a computer to received signals and control components. To facilitate this communication, we will use LabVIEW, a software package produced by National Instruments.

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a **visual programming language** (VPL) in that the programmer manipulates graphical elements to design a program instead of using text. Further, LabVIEW programs are intended to provide computer-based functionality that mimics that of a real, physical elecotronic instrument in a laboratory. For this reason, LabVIEW programs are referred to as **virtual instruments**, or VIs.

A classic example of such a virtual instrument is an oscilloscope, depictd in Fig. 1, which allows measurement and display of constantly varying voltages over time. The left oscilloscope is a physical instrument, with its display panel showing a plot of voltage over time. To the right is a LabVIEW implementation of a virtual oscilloscope. The virtual oscilloscope has knobs and buttons like the physical instrument and a live plot, but the "electronics" of the virtual instrument are graphic computer code behind the front panel with the knobs and display.

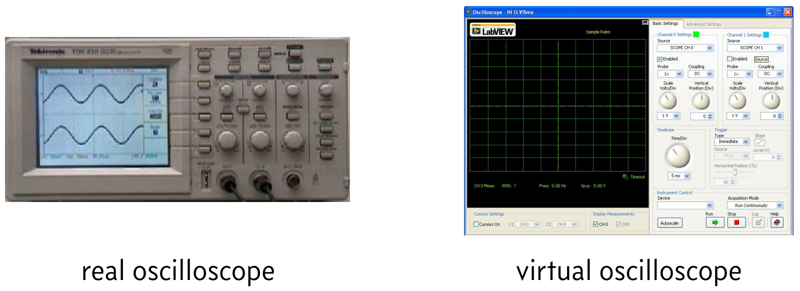


Figure 1: Left, a physical oscilloscope. Right, a virtual oscilloscope.

1.1 VI components: Front Panel and Block Diagram

When you open a new VI, which you can do by selecting New VI from the File pulldown menu, you will get two windows, one in back labeled Block Diagram and on in front labeled Front Panel. Fig. 2 shows the front panel and block diagram for a VI that generates a waveform.

As the name suggests, the front panel is the interface of the VI. Like any computer program, the interface takes inputs and displays or generates outputs. In LabVIEW speak, inputs are called **controls** and outputs are called **indicators**.

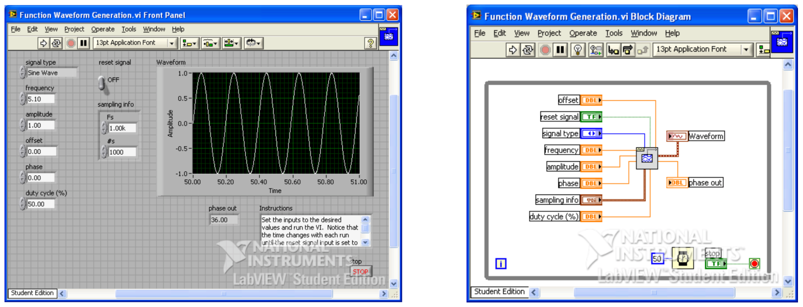


Figure 2: Left, a front panel of a waveform generator. Right, a block diagram of an waveform generator.

The block diagram contains the guts of the program. When you create controls and indicators in the front panel, the corresponding components appear in the block diagram as colored objects called **terminals**. The block diagram contains arithmetic operations, functions, constants, subVIs (akin to a submodule). The inputs and outputs of these objects flow through **wires**, which connect the object.

In my typical workflow, I build the front panel first. This is what I want my instrument to *do* and how I want to be able to control it. If you are used to programming in other languages, you can think of front panel design as design of your API, which you typically do first.

1.2 An example VI: Fahrenheit to Celsius converter

This is all rather abstract, and perhaps a bit complicated since a waveform generator is not the simplest VI we could imagine. Consider now a VI for converting an inputed number in degrees Fahrenheit to degrees Celsius, shown in Fig. 3.

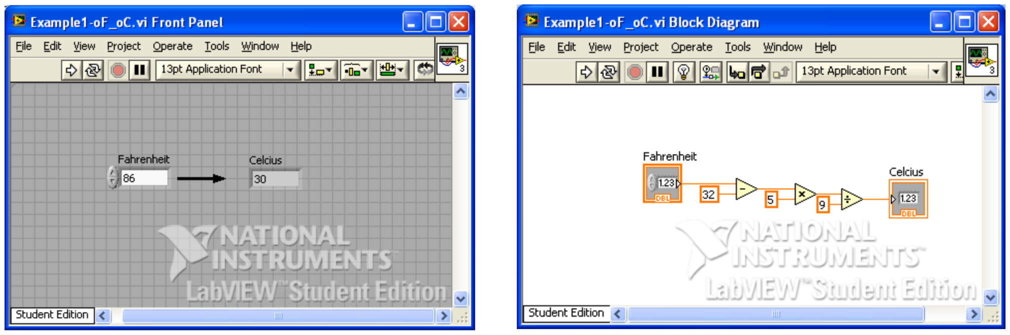

Example1-oF_oC.vi Front Panel
Fahrenheit
86
Celcius
30
Example1-oF_oC.vi Block Diagram
Fahrenheit
32
5
9
Celcius

Figure 3: A VI for Fahrenheit to Celsius conversion.

First, let's consider the front panel. We have a control (or input) in which the user specifies a number corresponding to the temperature of interest in degrees Fahrenheit. There is an indicator (or output) that gives the same temperature in degrees Celsius.

Looking at the block diagram, we see an input of data type DBL, or double. To convert to Celsius, we use the following formula.

$$C = \frac{5}{9}(F - 32). \tag{1.1}$$

So, we first subtract 32 from the inputted degrees Fahrenheit. This is accomplished with the minus mathematical operator. It takes two inputs, shown by the two orange wires to the left of the minus operator, and subtracts the bottom input from the top. So, the wires carry variables into operators. Coming out of that minus operator is the resulting difference of the two inputs. This then flows into a multiplication operator, which also takes two inputs, and then multiplies them together. We need to multiply by five, so this is the other input. The output is then divided by nine and delivered to the Celsius indicator.

1.3 The controls palette

You can add controls and indicators to your front panel using the **Controls Palette**. If it is not already in view, you can make it visible by selecting View → Controls Palette. An example annotated front panel is shown in Fig. 4, and an example controls palette is shown in Fig. 5. To add a control or indicator to the front panel, simply click on the icon in the controls palette and drag it onto the desired space in the front panel.

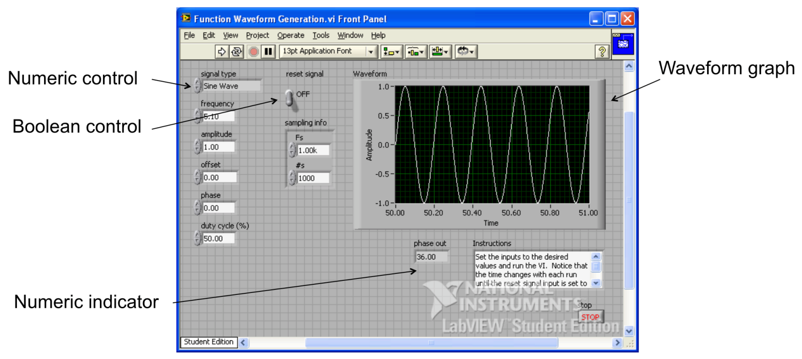


Figure 4: An annotated front panel for a waveform generator with various controls and indicators.

1.4 The functions palette

As the controls palette is your main resource for designing your front panel, the **Functions Palette** is your main resource for designing your block diagram. An annotated functions palette is shown in Fig. 6, and an example functions palette is shown in Fig. 7. The block diagram has a while loop and a subVI, both of which we will cover

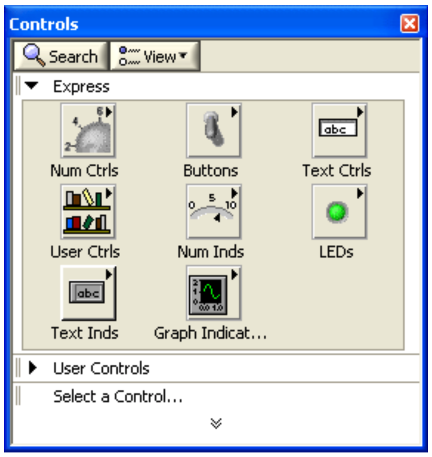


Figure 5: An example controls palette.

in coming lab sessions. The functions palette shows a subset of the many functions available.

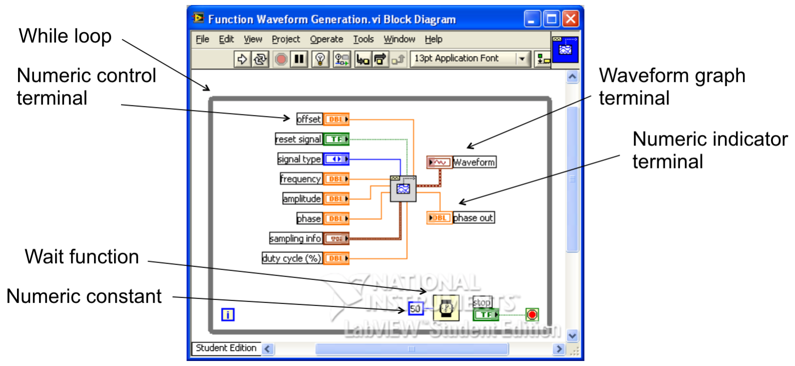


Figure 6: An annotated block diagram for a waveform generator. In the center is a subVI, which we will learn about in coming days.

1.5 The tools palette

The **Tools Palette** is useful for selecting and connecting element of your VI that you drug in from the functions and controls palettes. The **wiring tool**, which looks like a spool of wire, is used to connect objects in the VI. The position tool (the arrow) is used to select objects and then to position them. The pointed finger tool is used to operate your switches, knobs, sliders, etc., on the front panel.

Importantly, by selecting the automatic tool selection, LabVIEW will infer which tool you want to use based on where your pointer is. This is quite useful and can save you some clicking.

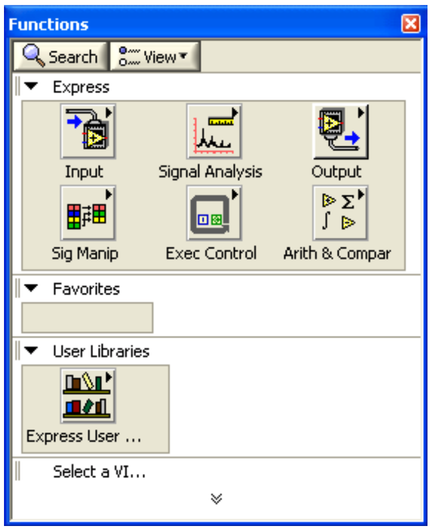


Figure 7: An example functions palette.

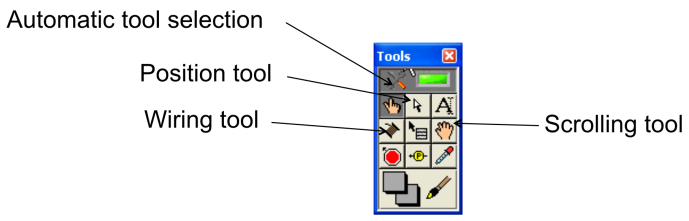


Figure 8: The annotated tools palette.

1.6 Tools to make your VI look pretty

Just as style is important in text-based programming, so too is it important in LabVIEW. Fortunately, LabVIEW offers several tools to prettify your VI. On the tool bar of both the front panel and the block diagram are alignment, distribution, and resizing tools for objects. You can spot them from the green and yellow colored boxes on their tabs. These tools work much as similar tools work in drawing/layout programs, such as Illustrator or PowerPoint.

You can also "clean" your wires, which can get bendy based on where the input and output nodes are by selecting a wire with the position tool and then right clicking and selecting Clean Up Wire. LabVIEW will then do its best to straighten and otherwise prettify the wire.

You can take a more, shall we say, fervent approach by clicking on the Clean Up Diagram button at the right of the toolbar of the block diagram window (the icon has a addition operator with a broom). This chooses an arrangement of objects and wires that is in some way optimal. Except for very simple VIs, I usually do not take this option because the rearrangements can sometimes be extreme. The result is straighter wires and convenient spacing, but my reasoning for how I set up the block diagram is destroyed.

1.7 Running your VI

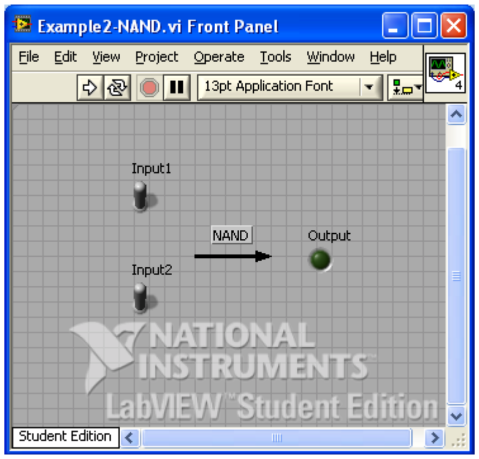
After you have built a VI, you want to **run** it. To run a VI once, click the forward white arrow on the toolbar of the front panel. Alternatively, you can hit Ctrl+R or select Operate → Run.

LabVIEW has the useful capability of running continuously. That is, it will listen for a change in a control, such as the user changing the value of a slider, and then change the indicator on the fly. To run continuously, click the two cycling arrows on the toolbar of the front panel.

You can abort execution by clicking the stop sign on the front panel toolbar, or pause executing by clicking the pause button.

1.8 A little practice

Before diving into your first homework, you might want to practice making a couple VIs. For your first practice, make a front panel with a couple of vertical toggle switches that you can flip up or down. Add an LED indicator that turns on in all instances except when both switches are up (a NAND gate). The VI is shown in Fig. 9.



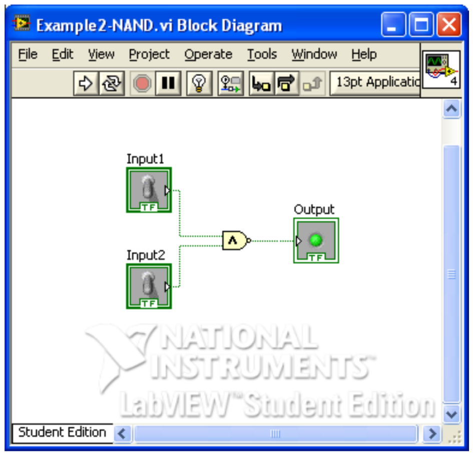


Figure 9: A VI for a NAND gate.

For your second practice, implement the Fahrenheit to Celsius converter in Fig. 3.

2 Data types, cases, and subVIs

In this lesson, I will present a hodgepodge of topics to help you gain more capabilities with LabVIEW.

2.1 Data types

Whether you are programming in LabVIEW or pretty much any other language, you will be working with variables. The following can be properties of a variable:

1. The type of variable. E.g., is it an integer, like 2, or a string, like 'Hello, world.'?
2. The value of the variable.

LabVIEW has many data types. For now, we will focus on three data types, **numeric**, **string**, and **Boolean**.

1. **Numeric** data are integers and floats. For controls (remember, "control" is LabVIEW speak for "input", which you put in the front panel), the numeric data type may be specified by right clicking the object, selecting Representation, and then choosing among the data types. There are many integer types, e.g., U16 is an unsigned 16-bit integer. Floats can be double (DBL, the default), or single (SGL). You may also specify extended precision (EXT), which depends on your hardware and operating system. LabVIEW also has capabilities for complex numbers, such as CDB for a complex double.
2. **Strings** are an array of characters. Importantly string comparison in LabVIEW is done character-by-character comparing the ASCII value of the characters. E.g., abc < abcd < e. Identical strings are considered equal.
3. **Booleans** take on values of TRUE or FALSE. Internally, they are stored as 8-bit values. Anything that is nonzero and cast as Boolean evaluates TRUE.

These basic data types can be organized into higher order structures, such as **arrays** and **clusters**, which we will talk about in coming lessons.

To convert form one data type to another in the block diagram of a VI, you can use the operators in the Programming → Numeric → Conversion pane of the Functions palette.

2.2 Useful quick-keys

As a graphical language, you will be pointing and clicking with your mouse a *lot*. It is nice to know a few quick keys.

Ctrl-S	Save a VI
Ctrl-R	Run a VI
Ctrl-E	Toggle between the front panel and block diagram
Ctrl-H	Toggle the Context Help window
Ctrl-B	Remove all bad wires
Ctrl-W	Close the active window
Ctrl-F	Find objects or text
Ctrl-Tab	Cycle through LabVIEW windows

These are just a few. You should check out this reference guide from the National Instruments website.

2.3 Debugging tools

As with any code, you are likely to have bugs in your LabVIEW VIs. Fortunately, LabVIEW has some debugging tools to help you find and fix your bugs. If there is the graphical programming equivalent to a syntax error, that is an error that prevents LabVIEW from compiling your VI, the normally white, intact Run arrow will be gray and broken, as shown in Fig. 10. If you click on it, you will get an Error list, which will show descriptions of the errors in your front panel and block diagram. If you click the Show Warnings checkbox, you will also see warnings that do not necessarily prevent compilation, but are bad programming practice and can lead to bugs.

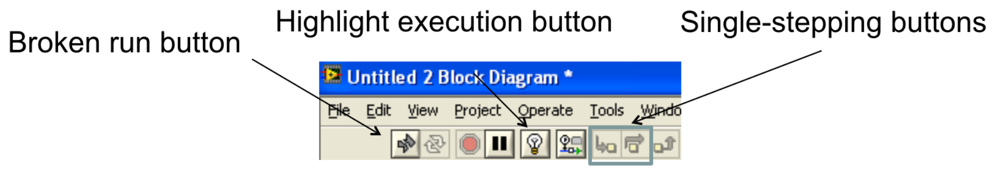


Figure 10: A block diagram toolbar with debugging tools highlighted.

You click the Highlight execution button (a lightbulb) to visually show the VI execution. This is much like a standard debugger and reduces performance. Finally, you can select single-step modes to step into or step over a node in a black diagram.

You can insert breakpoints into your code using the Breakpoint button on the Tools Palette (Fig. 11). You can also insert a probe, allowing you to observe variable values while your VI is running.

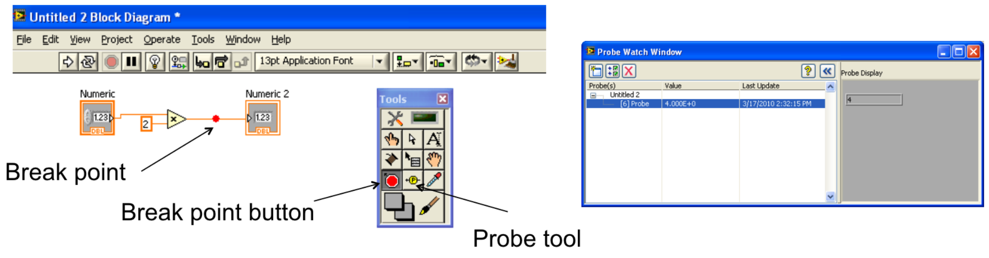


Figure 11: Left, illustration of break points and probe tools. Right, example probe watch window.

2.4 Case structures (LabVIEW's *if else if*)

Case structures allow you to implement *if else if*-type statements in LabVIEW. They behave like *switch case*s in C and Java. Their graphical representation is illustrated in Fig. 12.

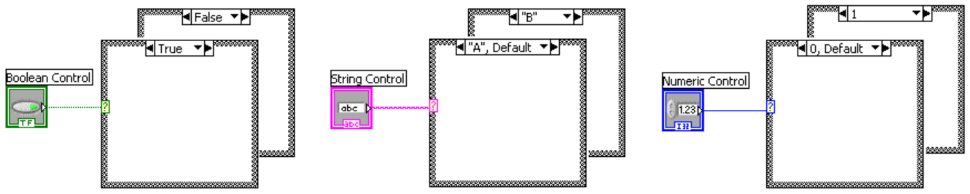


Figure 12: Case structures with different types of input.

Let us consider first the case structure that takes a Boolean as input. There are then only two possible cases, TRUE or FALSE. To enter objects into the case structure, you need to place objects and wires in the box. To toggle between what operations happen in the TRUE case and the FALSE case, you click on the menu at the top of the case structure box.

We can also input a string or Enum control (a ring-style control—remember "control" is LabVIEW speak for "input"). If this type of control comes into the case structure, you are unrestricted in the options for the cases, i.e., you are no longer restricted to either TRUE or FALSE. Therefore, you must specify a default case (akin to the else clause of an if else if statement).

Finally, you can have numeric control. LabVIEW has strange behavior for numeric control and, in my opinion, you are asking for bugs if you directly input numeric control. I will therefore not comment any further on this, and just advise you not to do it.

2.5 Modular programming: SubVIs

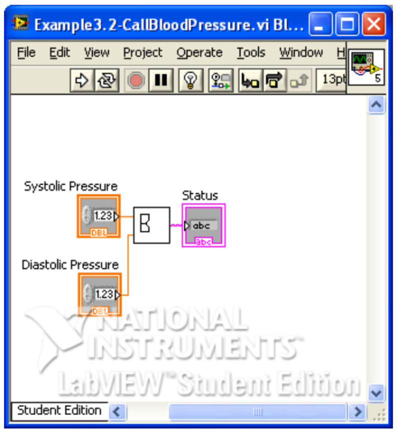
When you write text-based code, you often write functions that do specific, small tasks. They have a prototype; they takin input, perform operations on it, and then return output. You larger scale program or project calls these functions in succession, perhaps with logic and looping. This is generally good practice, to do **modular programming**.

The same is true for LabVIEW. *Any VI you write can be incorporated into another VI.* When a VI is used in another, it is called a **subVI**. When building your LabVIEW programs, you should create many small VIs that perform well-defined takes that are simply and clearly connected in your main VI.

To have a concrete example in mind, consider a VI that evaluates blood pressure measurements to give a categorical characterization. The category of blood pressure is given in the table below, where the more severe category is chosen.

Systolic		Diastolic	Category
< 120	AND	< 80	Normal
$120 - 139$	OR	$80 - 89$	Prehypertension
$140 - 159$	OR	$90 - 99$	Stage 1 high blood pressure
≥ 160	OR	≥ 100	Stage 2 high blood pressure

The block diagram for the main VI for this program and that for the subVI are shown in Fig. 13. This example is somewhat trivial in that the subVI is the entirety of the routine, but serves to show how subVIs can be situated in a VI.



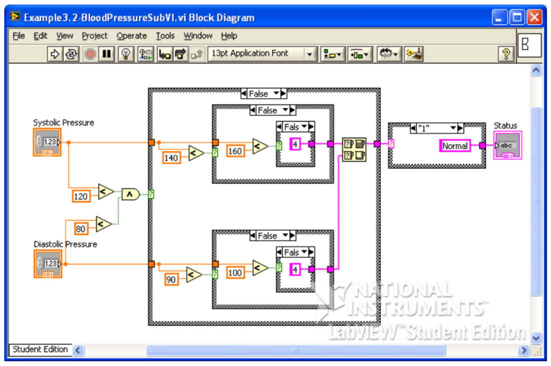


Figure 13: Left, block diagram of top level VI for a categorial indicator of blood pressure. Right, the block diagram for the subVI at the heart of the program.

To make your existing VI capable of being inserted as a subVI, you need to take a few steps working on the front panel of your VI. In the upper right corner of the

VI is an icon that looks like an an oscilloscope with an addition operator. This will be the icon representing your subVI when you insert it in another VI. To edit this, right click on the icon an select Edit Icon.... You can then edit the icon with rather primitive and annoying tools. I usualy just make the icon a box with text in it.

After choosing your icon, you need to specify how your subVI connects to controls and indicators (inputs and outputs). To do this, right click on the icon and select Show Connector (step 1 in Fig. 14). Click on the connector, which not replaces the icon. (On newer versions of LabVIEW, the connector is always shown next to the icon.) Right click the connector and select Patterns, and a palette of connector patterns appears. The default pattern is a $4 \times 2 \times 2 \times 4$ This allows for four inputs to the left and four outputs to the right. Some LabVIEW users advocate for always using the default, even if you do not need that many inputs and outputs. I advocate against this because this is poor programming practice: you should prototype your subVI so that you know what inputs and outputs to expect. You should pick the input/output pattern that is appropriate for your function. Next, use the wiring tool to select which controls and indicators in your front panel correspond to which inputs and outputs in your connector pattern (steps 3 and 4 for Fig. 14). After saving, you will have a subVI that you can plop into any other VI you are making. The inputs and outputs behave like the controls and indicators of the front panel of the original VI.

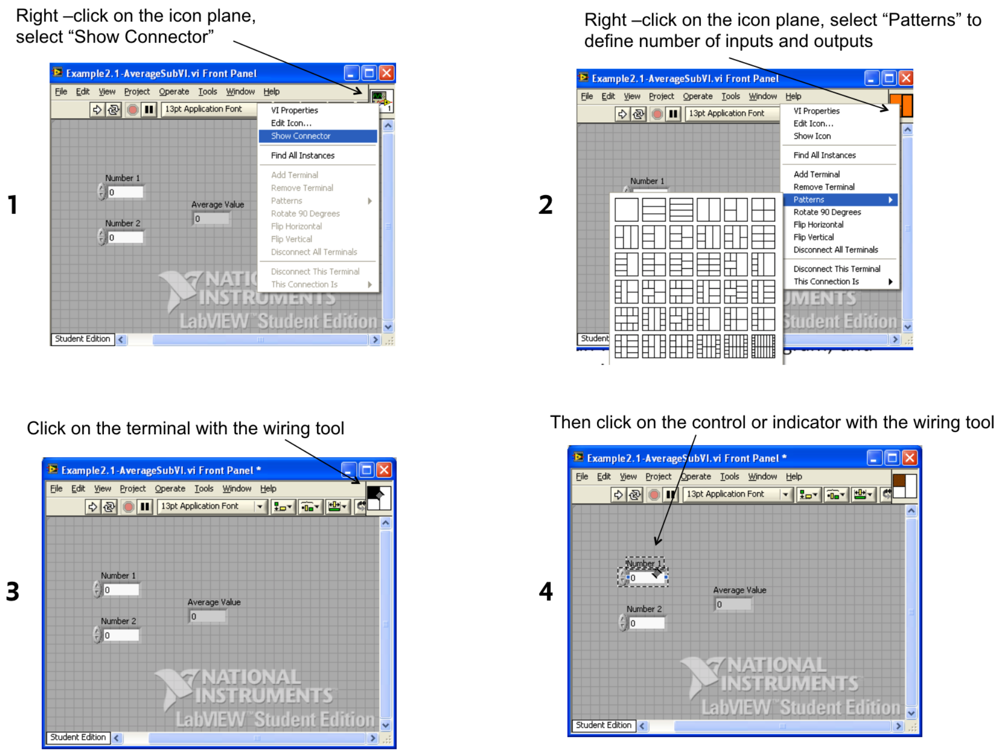

Right –click on the icon plane, select “Show Connector”
Right –click on the icon plane, select “Patterns” to define number of inputs and outputs
1
2
Click on the terminal with the wiring tool
Then click on the control or indicator with the wiring tool
3
4
Example2.1-AverageSubVI.vi Front Panel
VI Properties
Edit Icon...
Show Connector
Find All Instances
Add Terminal
Remove Terminal
Patterns
Rotate 90 Degrees
Flip Horizontal
Flip Vertical
Disconnect All Terminals
Disconnect This Terminal
This Connection Is
Number 1
Number 2
Average Value

Figure 14: The steps to enabling a VI to be a subVI.

To add a subVI to a block diagram you are working on, scroll to the bottom of the Functions Palette and click on Select a VI.... You will then be able to wire it up according to your specification of the connector pattern.

To get an overview of how all of the subVIs are connected and depend on each other in a top level VI, click on View $\rightarrow$ VI Hierarchy.

3 Arrays, clusters, and plotting

3.1 Clusters

We saw last time that more than numbers can flow through wires in a VI. We made **error clusters** and we propagated through our VI. I did not carefully define what a **cluster** is. A cluster is a grouping of objects, must like a structure in C or a dictionary in Python.[1] When we make an error cluster, we specify an error code and an error message, and optionally whether the error is a warning and whether or not to show the full chain when the error is encountered, which are updated in an existing error cluster. Conveniently, we can run the entire cluster through wires in the diagram.

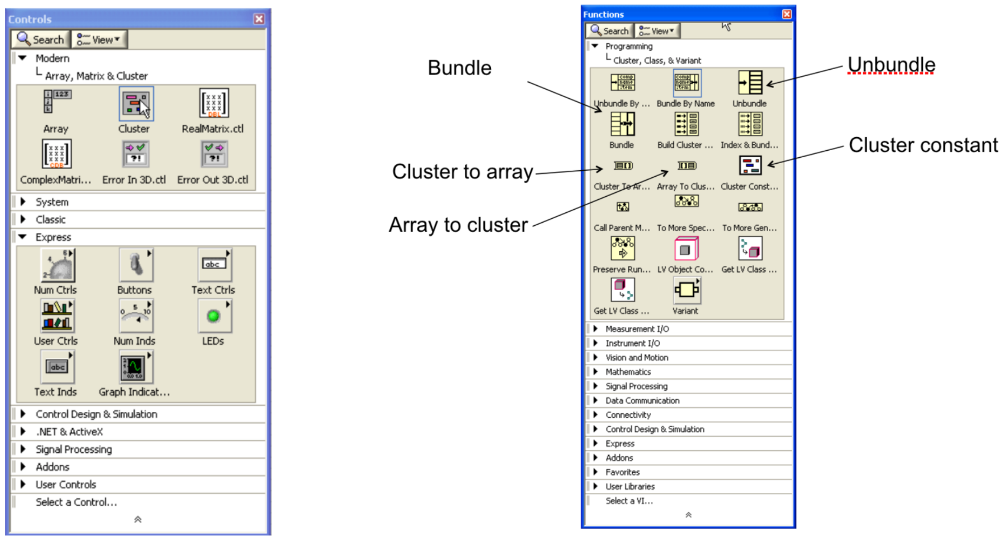


Figure 15: Left, controls for clusters for the front panel. Right, annotated cluster functions from the Functions Palette.

We can make clusters beyond error clusters. To insert a cluster in the front panel of a VI, choose Cluster in the Array, Matrix & Cluster menu of the Controls Palette (see Fig. 15). Once you drag this onto the front panel of your VI, you can drag whatever in controls you want to bundle into this cluster.

In the back panel, we can unpack clusters using the Unbundle object from the Functions palette (see Fig. 15). We can also bundle objects together into a new cluster using the Bundle object in the same palette.

To see an example use of clusters, see the `mass_of_air.vi` sample VI that computes the mass of air given the pressure, volume, and temperature using the ideal gas law.

[1]A LabVIEW cluster is actually more like a namedtuple from the collections module of the Python Standard Library. A LabVIEW cluster is not a hash table like a Python dictionary.

This tutorial is useful to learn about both clusters and arrays.

3.2 Arrays

Arrays are similar to clusters in that they are ordered collections of objects. They differ in that all of the objects must be of the same type. Further, arrays may be multidimensional, not just an ordered one-dimensional array. Because all of the data are of the same type, more operations may be done on arrays.

Arrays are **indexed**, as in other languages. Importantly LabVIEW array indexing starts at zero. You can index an array using that Index Array object in the Functions → Programming → Array palette (see Fig. 16).

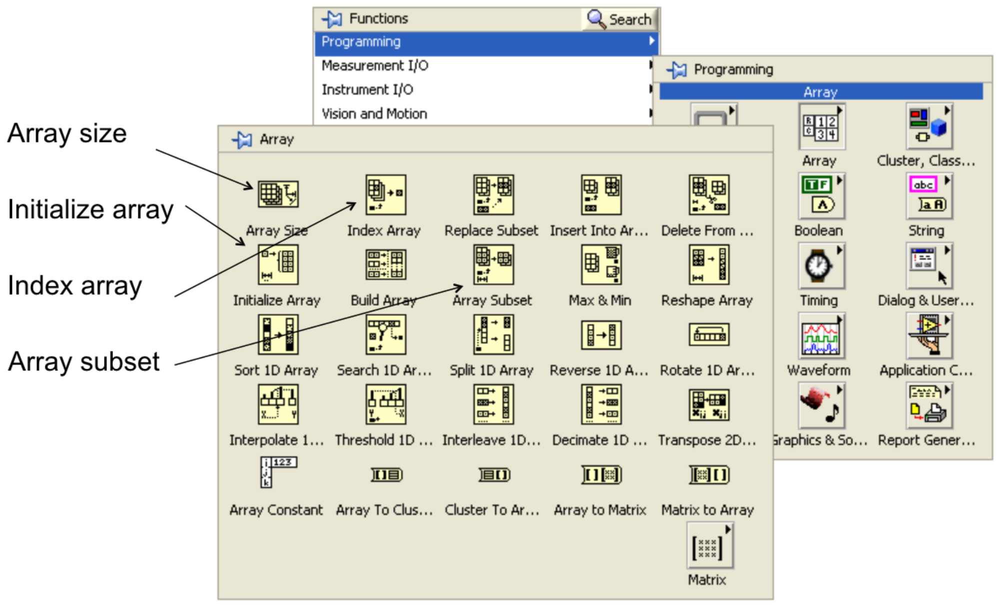


Figure 16: Annotated palette of array functions.

To create an array in the front panel, drag an Array object onto your front panel, as in Fig. 17. You will then need to populate it with a numeric constant or a strong constant to specify the data type of the array. When it appears on your front panel, the array will have an index selector to the left and then the values of the entries of the array to the right. If you want to make a two or more dimensional array, you can right click on the index selector and select Add Dimension. You can do a similar procedure by adding an array constant to a block diagram.

The array function palette (Fig. /16) has lots of options for building and modifying arrays. For an example of using the Insert Into Array function to split an array into two arrays, one containing the nonnegative values in the original and the other containing the negative ones, see `pos_neg_array.vi`.

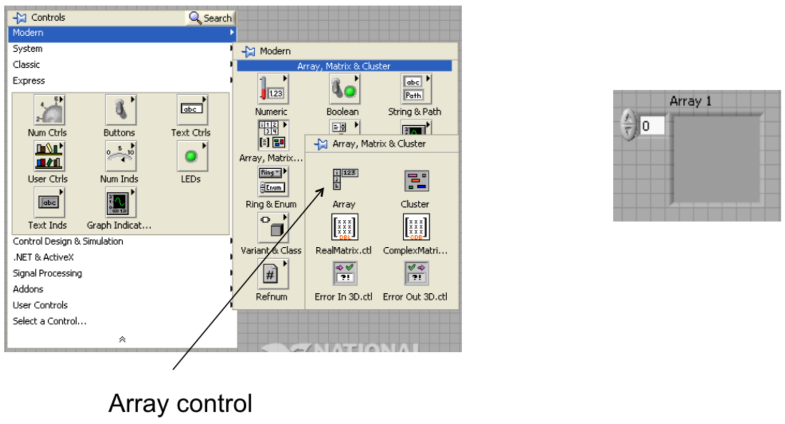


Figure 17: How to add an array control.

3.2.1 Operations on arrays

Arithmetic operations on arrays are elementwise. For example, if you add two arrays by wiring them into a + operator, the result is an array where like-indexed input values are added together. If you add an array and a scalar, the scalar value is added to each element of the array, with the result being an array. Similar results happen for other operations.

What happens when you add two arrays of different lengths? The result is an array equal to the length of the shorter array. I personally think this is a terrible idea. It just invites bugs. Therefore, *Beware*. You might see unexpected behavior in your VIs because of this. You might want to write your own VI that does error checking for array operations with arrays of different sizes, such as in the example `add_arrays_with_error.vi`.

3.2.2 Matrices

What if instead of doing elementwise multiplication of two 2D arrays (where one is $m \times p$ and the other is $p \times n$), I want to do a matrix multiplication? We can still use a two arrays and use the A $\times$ B function in the linear algebra palette (see Fig. 18).

Alternatively, I could convert the arrays into a new data type called a **matrix**. The $\times$ operator then acts like matrix multiplication on these objects. In my view, this is another bad idea. Use of the A $\times$ B function is unambiguous; it is completely clear that you are attempting matrix multiplication. Matrices and arrays are stored in the same way for LabVIEW, and there is no performance boost for using matrices versus arrays. I am of the opinion that you should never use matrices. The added flexibility is not helpful; it just opens you up to hard-to-find bugs. You can do all the matrix calculations you need using arrays, and matrices were not even added to LabVIEW until version 9.0 (I think).

3.3 Polymorphism in LabVIEW

Polymorphism is generally the idea that a given interface, say to a function, accepts inputs of different types and then does operations on them according to the type. One way this is achieved in Python, for example, is by operator overloading.

```
3 + 4 = 7
```

```
'base' + 'ball' = 'baseball'
```

The Python + operator can work on data of different types. LabVIEW also features polymorphism. The LabVIEW + operator can take different data types, as we have seen; it can take scalars or arrays, or a mix (or other types as well, including clusters; check out the sample VI `array_cluster_polymorphism.vi` to see how array operations can work on clusters). Usually, the choice of operation based on input type is intuitive, but not always, as we saw in the example of two arrays of different lengths. (I would expect an error to be raised.)

I bring up this polymorphism because it is at once convenient and dangerous. Your code might work in unexpected ways depending on the types of inputs you give. *Caveat emptor.*

3.4 Making *x*-*y* plots

LabVIEW's plotting applications are primarily built for real-time monitoring of signals. After all, you are creating virtual instruments. As a result, its plotting capabilities are limited, but nonetheless useful.

Plots are shown in the front panel, usually as indicators. To make a plot of *x*-*y* data, I usually use the Ex XY Graph VI from the Graph palette of the Controls palette. Dragging this onto your front panel will create the graph, along with an object on the back panel. By right clicking on the graph, you can set properties, such as how the data are represented (points versus lines, etc.).

To see an example of the construction of an *x*-*y* plot of one of my favorite data sets (relative levels of Bicoid protein across a *Drosophila* embryo from the classic Driever and Nüsslein-Volhard paper), see `plot_bcd_profile.vi`. Note that you wire in an array of *x*-values and *y*-values to specify the data points.

3.5 Waveform charts and graphs

More typically, you will want to monitor real-time signals that have evenly sampled points. To do this, you can use Waveform charts and graphs. To make a waveform chart or graph, drag the Waveform Chart or Waveform Graph onto your front panel. Once in the front panel, you can change the xaxis labels, as well as the plot title, which is to the top left. At the top right is the plot legend, which you can right click on to alter the appearance of the plot.

In the block diagram, a wave *chart* takes as input a scalar value. The plot is constructed as the value coming into the wave chart changes. This is in contrast to a wave *graph*, which takes an array of data points as input. Alternatively, you can construct a cluster that has the starting time, the time between samples, and the array containing the signal. This will scale the x (time) axis appropriately.

To see examples of a waveform chart and a waveform graph, look at `wf_chart_and_graph.vi`. Note that the waveform chart is inside the for loop, while the waveform graph is outside. An array is automatically created within the for loop, which is outputted to the waveform graph.

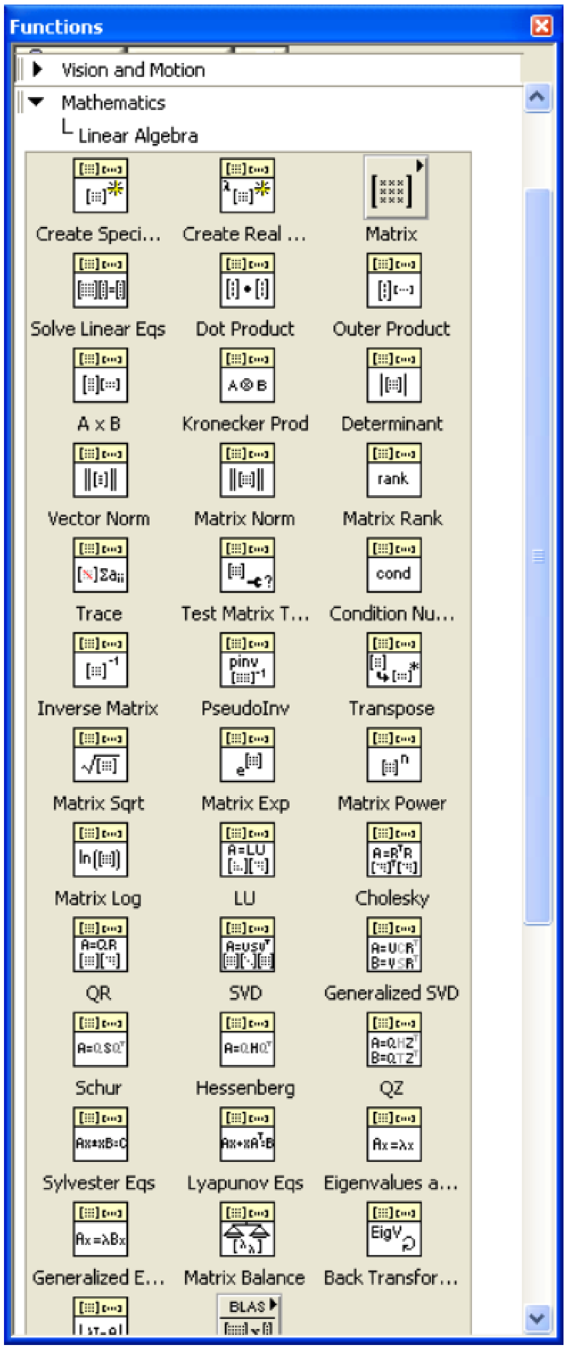


Figure 18: The linear algebra palette.