Introduction to Modern Data Acquisition with LabVIEW and MATLAB

By Matt Hollingsworth

Introduction to Modern Data Acquisition

Overview

Goal: To learn to use the various computer tools available to acquire and analyze experimental data.

Materials:

- Computer with LabVIEW and MATLAB
- NI USB-6008
- SummaSketchIII
- Conductive paper with electrodes
- Various cables
- Brain

This lab exercise is meant to give a sweeping overview of a couple of the tools available to a modern experimental physicist. The tools that this exercise will focus upon are LabVIEW and MATLAB.

In general, LabVIEW is a useful programming language when you need to produce some code that will acquire some data in a lab environment. You can quickly create code that will allow you to acquire some data, do some data analysis, display real-time results of that data, and export it in a format that will be capable of being read by other data analysis software (such as, in our case, MATLAB).

MATLAB, on the other hand, is a handy mathematical toolbox that comes with many features that are useful for data analysis. It is also a widely accepted industry standard, so LabVIEW comes with built-in support for directly interfacing with the script server for MATLAB. This allows you to input commands from LabVIEW directly into the MATLAB kernel and have them executed as if you were typing them in the MATLAB command window. We will be exploiting this feature in order to generate some detailed visualization of the electric field between two electrodes.

This paper will not but scratch the surface of the functionality available to you through the LabVIEW and MATLAB platforms; however, my goal is to make you conversant enough to where you know what questions to ask/documentation to search for and you can understand the answers that you are given. The more acute functionality is left for you to discover yourself as you need it.

Furthermore, if you already know something that I am discussing in the LabVIEW and/or MATLAB sections, feel free to skip it and move on to something else. There are no exercises or anything in those two sections; the most important thing is that you are able to perform the final task presented to you in the actual lab.

In case you are having trouble, each example has its corresponding .vi or .m file on my web site at http://www.evanescenthorizons.com/. To find the examples, look under Labs > Introduction to Modern Data Acquisition.

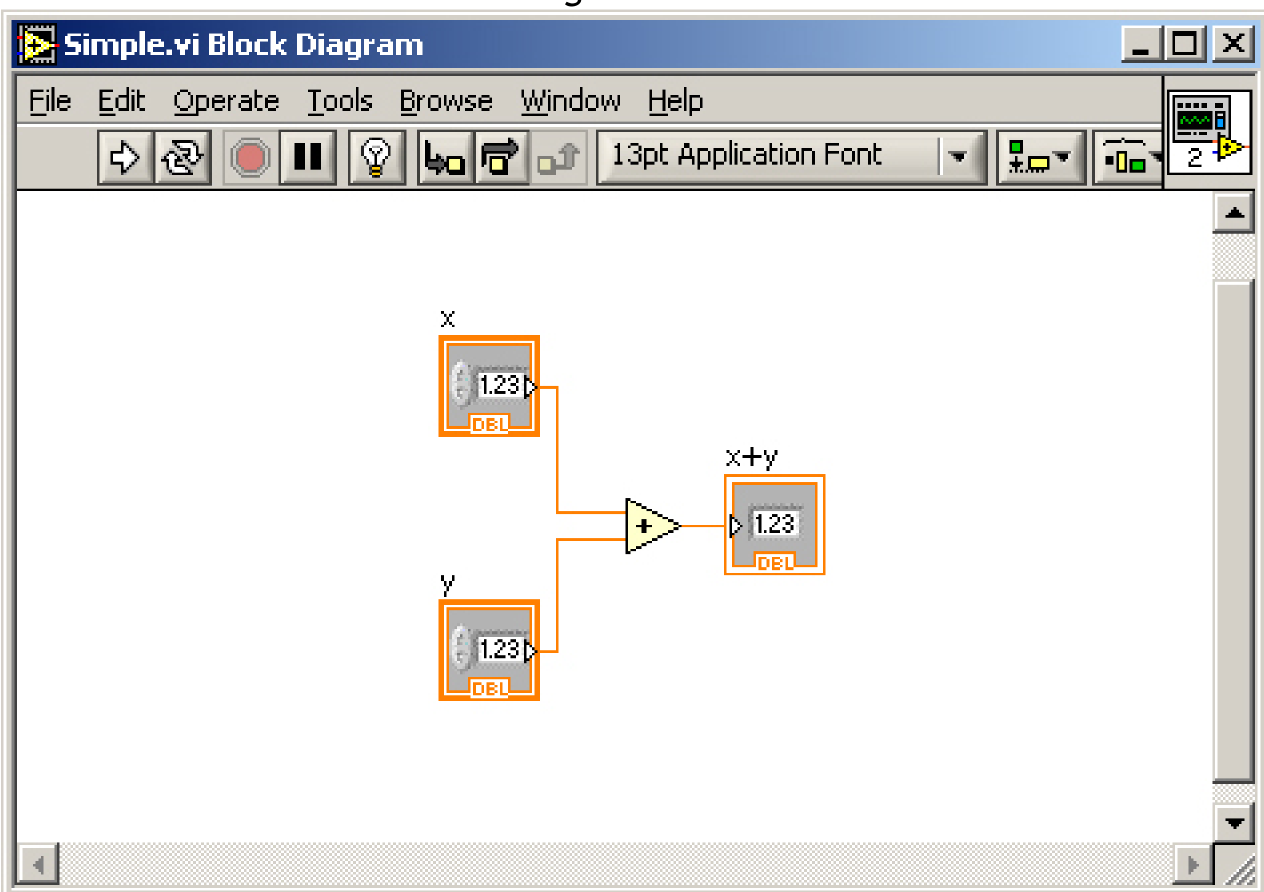
If you get stuck or have any questions, comments, complaints, or corrections, feel free to e-mail me at mhollin3@utk.edu. I'd be more than happy to help, and I welcome your feedback.

LabVIEW

Section 1.1: Introduction to LabVIEW

LabVIEW is a graphical programming language/IDE combination that is tailored for use in a lab environment. The basic analogy throughout LabVIEW is that of a virtual instrument or VI. In a LabVIEW program, just like a real instrument, you have controls (input), indicators (output), and logic to define the relationship between input and output. Here is a sample of a very basic LabVIEW program:

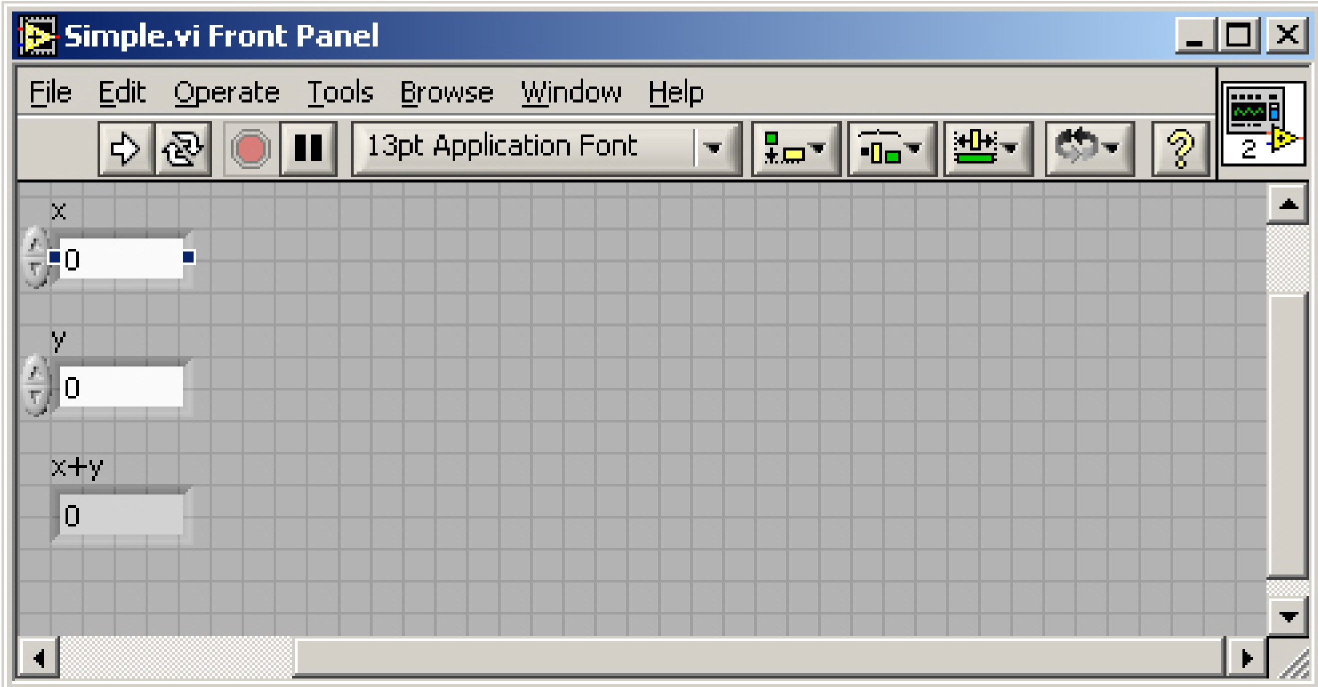
Figure 1.1.1



As you can probably already tell, this program takes two numbers, x and y, and outputs its sum. In LabVIEW, the lines that show data flow (in this case, they are thin orange lines, meaning they are single length, double accuracy numbers) are called wires. Inputs are called controls and outputs are called indicators. The plus sign inside the triangle is called a subVI, which is analogous to a subroutine or function in text-based languages. In this case, the sum subVI accepts two numbers as arguments and outputs the sum.

Logic and interface are two separate entities in a LabVIEW program. The interface is called the front panel, while the programming logic itself is called the block diagram. Here is the front panel for the program pictured above:

Figure 1.1.2



In order to run a program, you simply click the run arrow in the top left corner. To toggle looped execution, click the button with the revolving arrows (this causes the program to run continuously). The main program execution buttons are detailed below.

Figure 1.1.3 - Typical Execution Bar

: Run Button

: Run Continuously Button

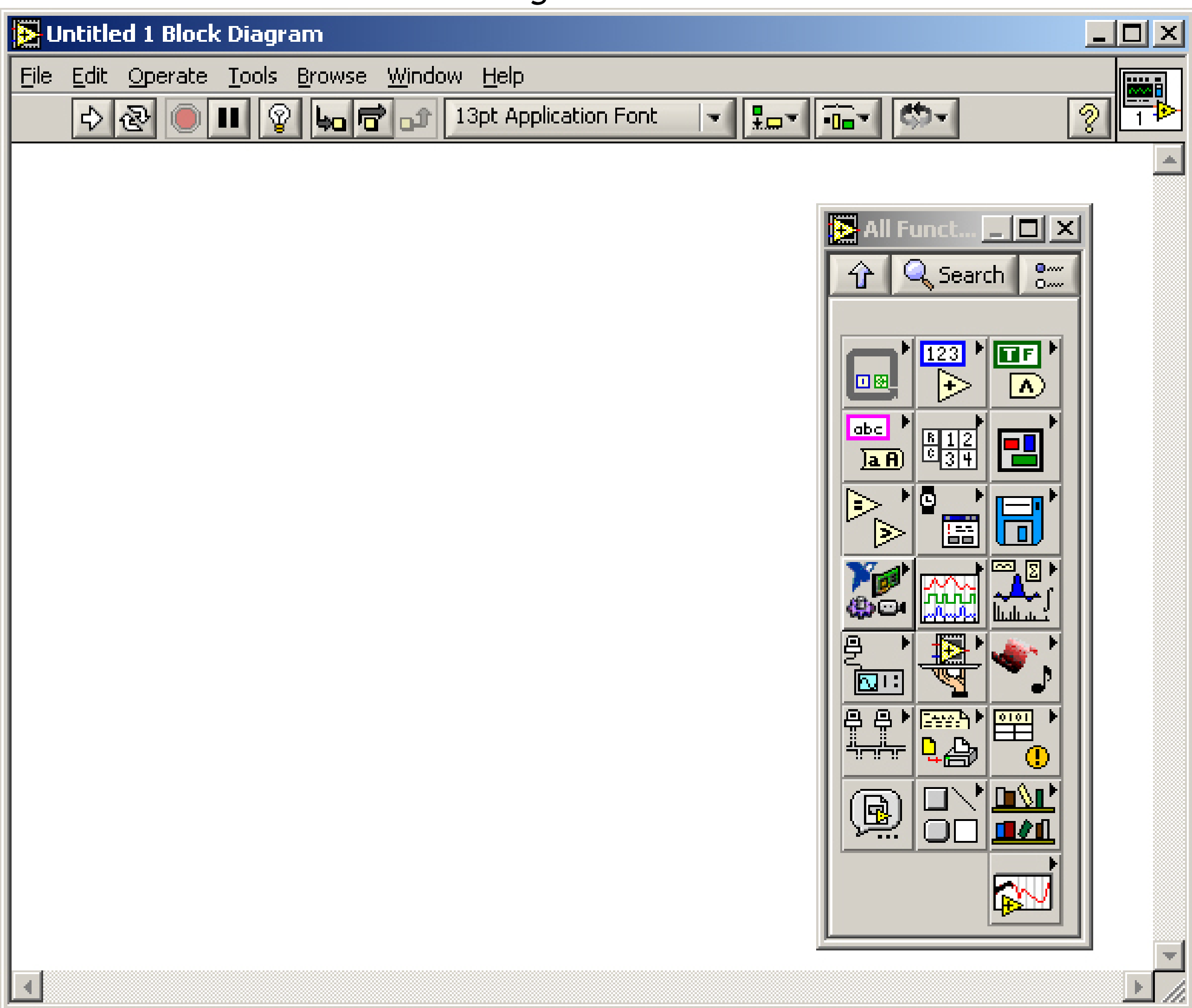
: Abort Button

: Pause Button

Section 1.2: A Simple LabVIEW Program

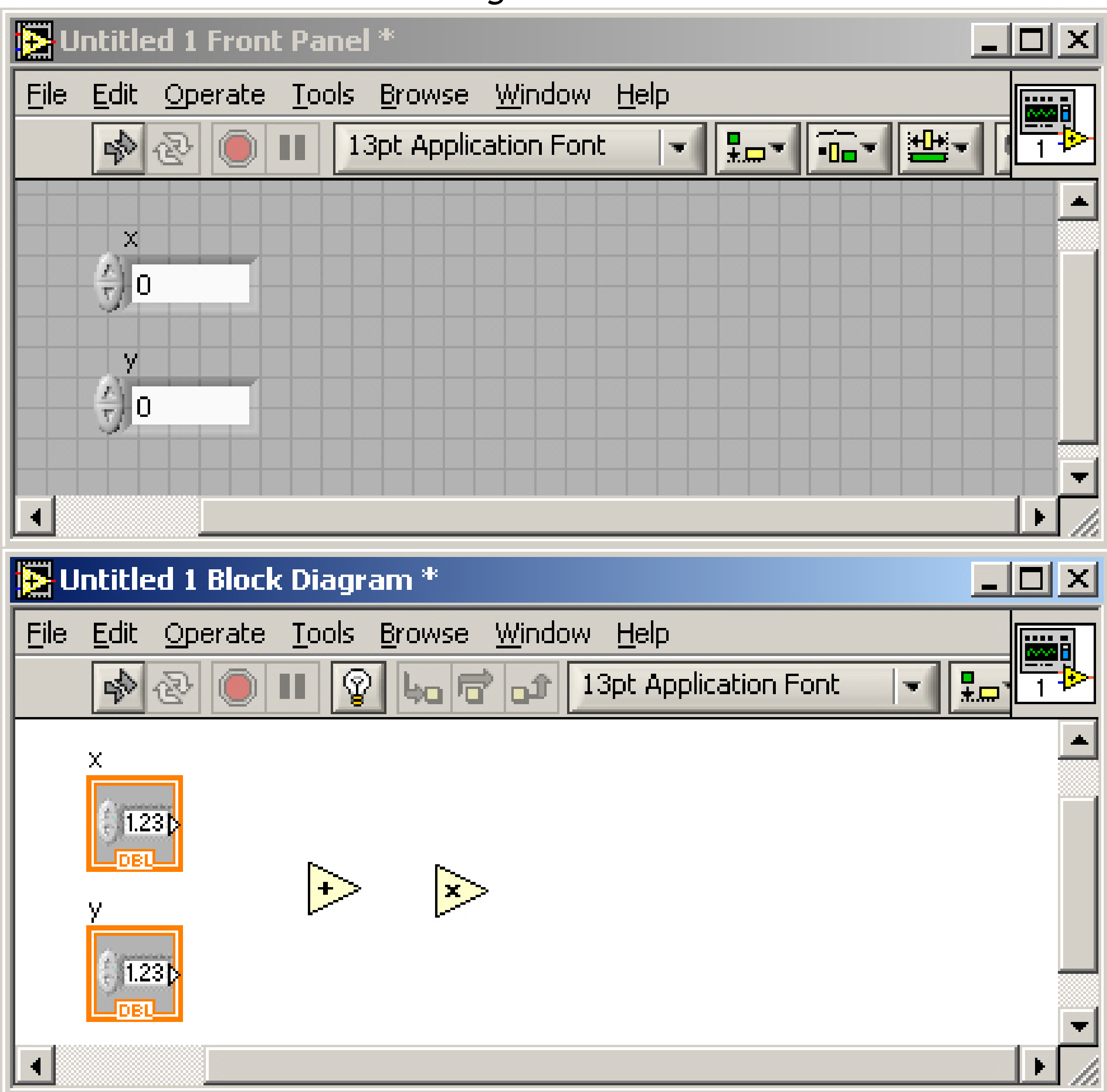
To begin writing a new LabVIEW program, simply start up LabVIEW, click new, and select blank VI when you are asked for a template. You now have a blank front panel and a blank block diagram. Navigate to the menu bar at the top of the screen, click window, and select "Show Block Diagram" (or press Ctrl + E). Right click anywhere in the white space of the block diagram to pop up the functions menu. Click "All Functions" in the bottom right of the popup window, and then click the thumbtack in the top right corner to keep the functions window open. Your screen should look something like this:

Figure 1.2.1



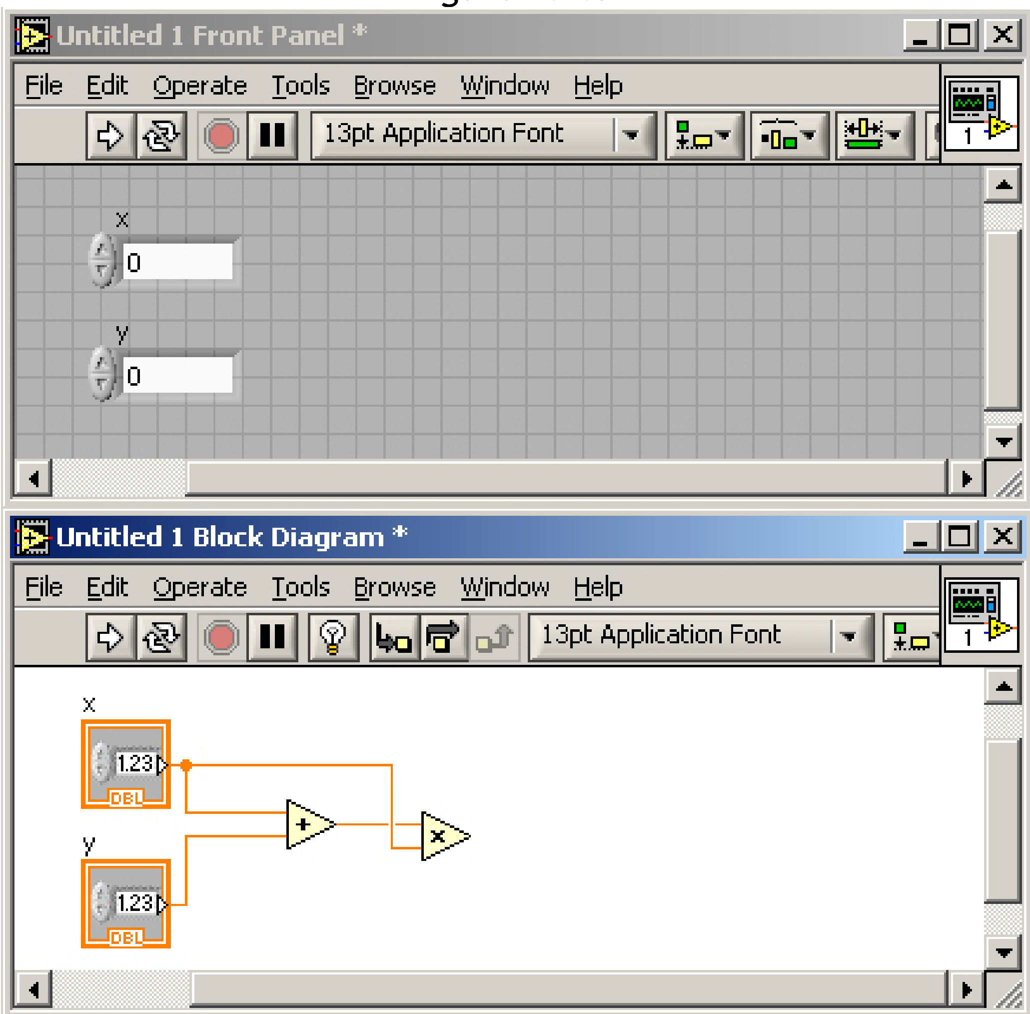
Click on the numeric box in the functions pallet (the on with the + and the 123 in it). Click and drag a sum subVI and a multiplication subVI onto the block diagram. Now we have some program logic, but we need something to input data. Navigate back to the front panel by selecting Window > Show Front Panel from the menu bar at the top. Right click anywhere on the front panel, select the "Num Ctrls" box, and click and drag a "Num Ctrl" onto the front panel. Repeat so that there are two numeric controls on the front panel. Double click on the label of one of the controls ("Numeric" by default) to change the name of the control to "x". Change the other to "y". After all is said and done, your front panel/block diagram should look something like this:

Figure 1.2.2



Now, we will tell LabVIEW to output z, where z = (x+y)*x. To do this, we wire x and y to the sum subVI, wire the output of sum to one of the inputs of multiply, and then wire x to the other input of multiply, like so:

Figure 1.2.3

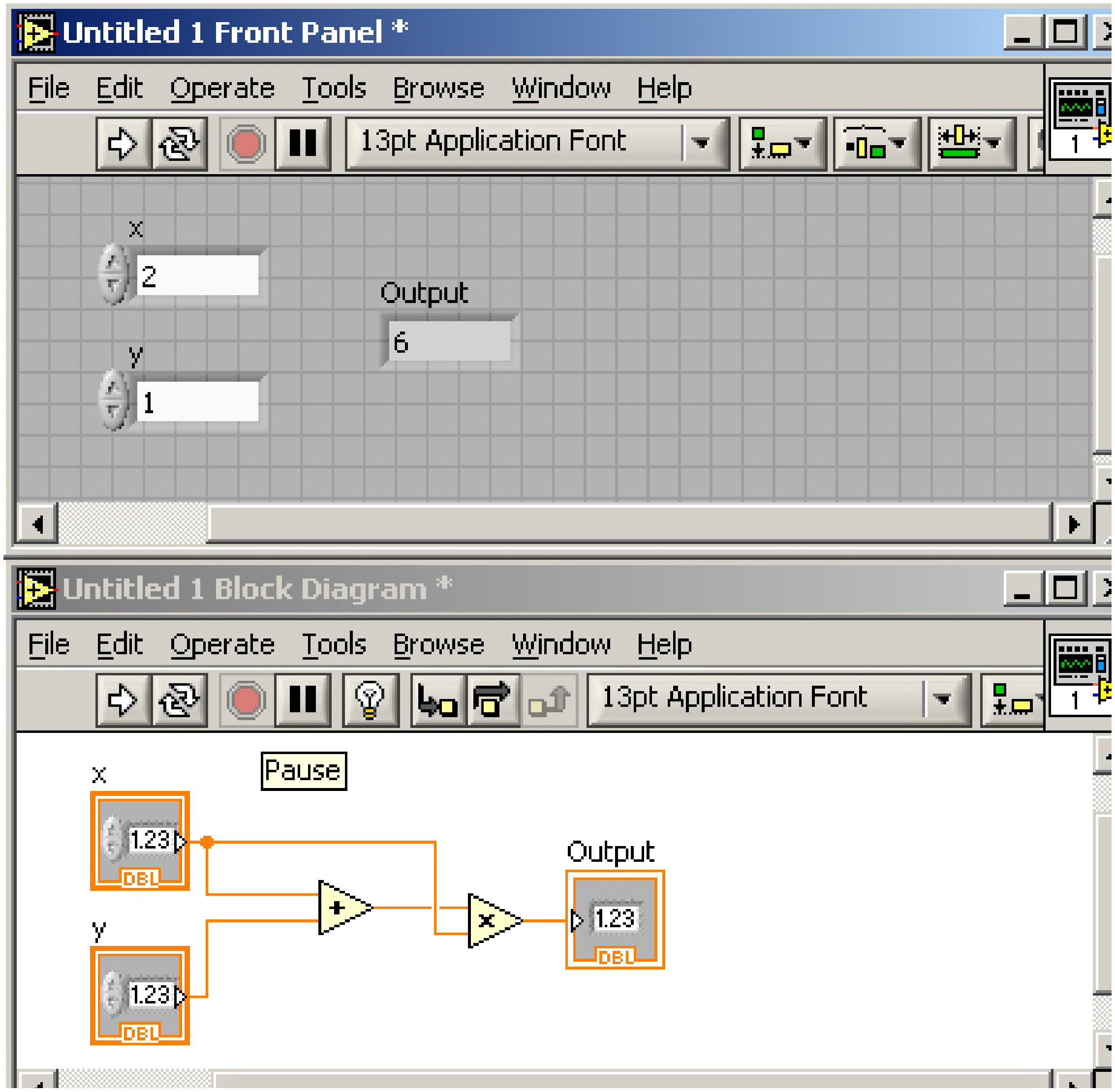


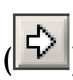
Side Note If you ever see wires appear as dotted lines with x's in various places on them, you wired something wrong. If this happens you may either right click the wire and tell it to delete the wire branch, left-click somewhere on the wire and finish wiring it properly, or hit Ctrl + B to remove all broken wires from the block diagram.

Now all that is left is to create an indicator. You can return to the front panel and create a numeric indicator the same way that you created the numeric controls, but there is a quicker way. Right click on the output of the multiplication subVI (the right most tip of the triangle). You'll see a menu pop up; select Create > Indicator. LabVIEW adds and automatically wires an indicator the output. For future reference, you may do the same thing when creating both indicators and constant terms.

Here is the completed VI:

Figure 1.2.4

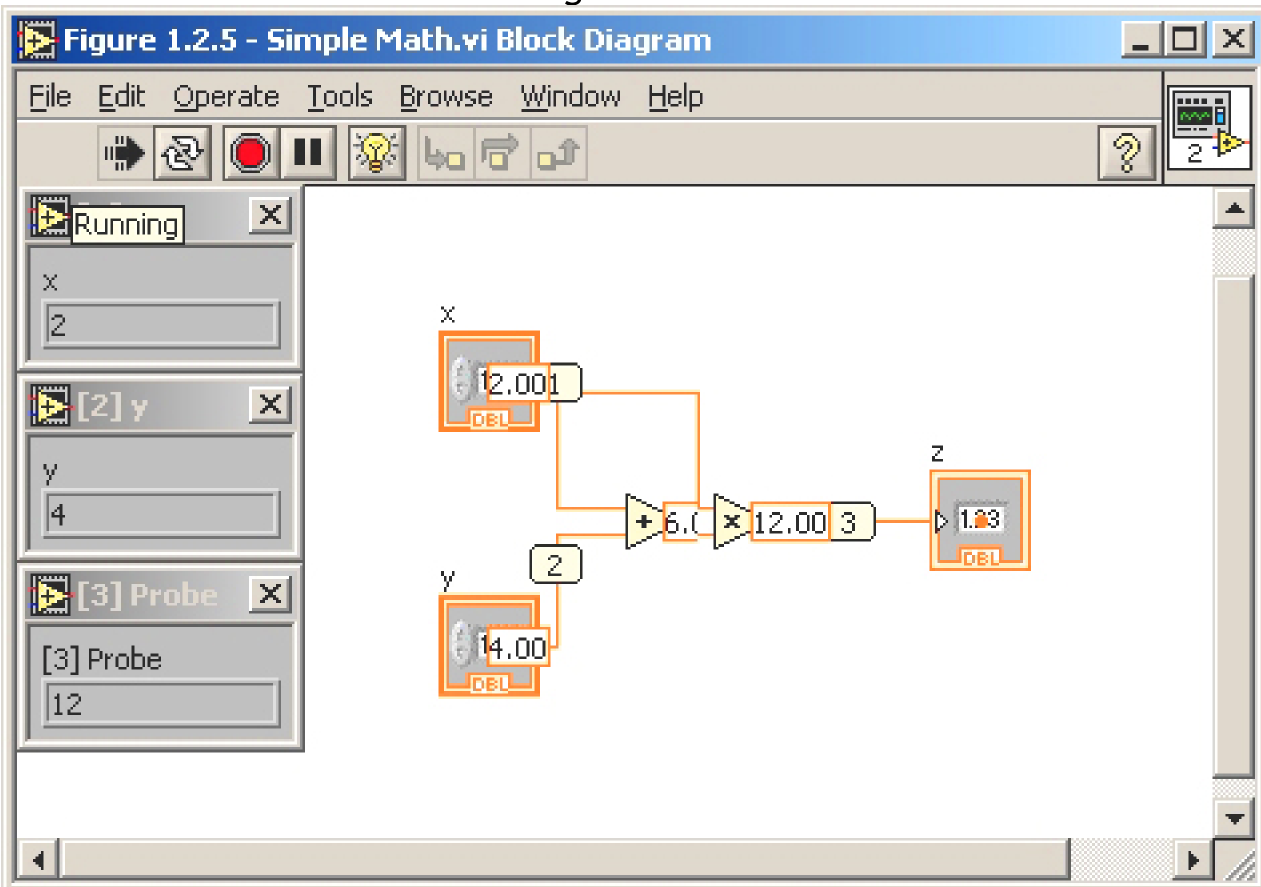


The program is now complete. Now, to test your new VI simply click the run button () to run your VI. Every time that you change x/y and run the VI, you should see an output, called "output" in our program, that corresponds to output = x*(x+y)

If your program is not functioning properly, you may always click the little light bulb that is by the pause button in the menu bar of the block diagram. When this button is toggled, "Highlight Execution Mode" is turned on. This allows you to see your program execute step by step—a very useful debugging feature.

Furthermore, you may go to Window > Show Tools Palette, and place a probe on a wire by clicking the  button and then clicking on a wire of which you would like to monitor the output. Here is what a probed VI with execution highlighting turned on looks like.

Figure 1.2.5



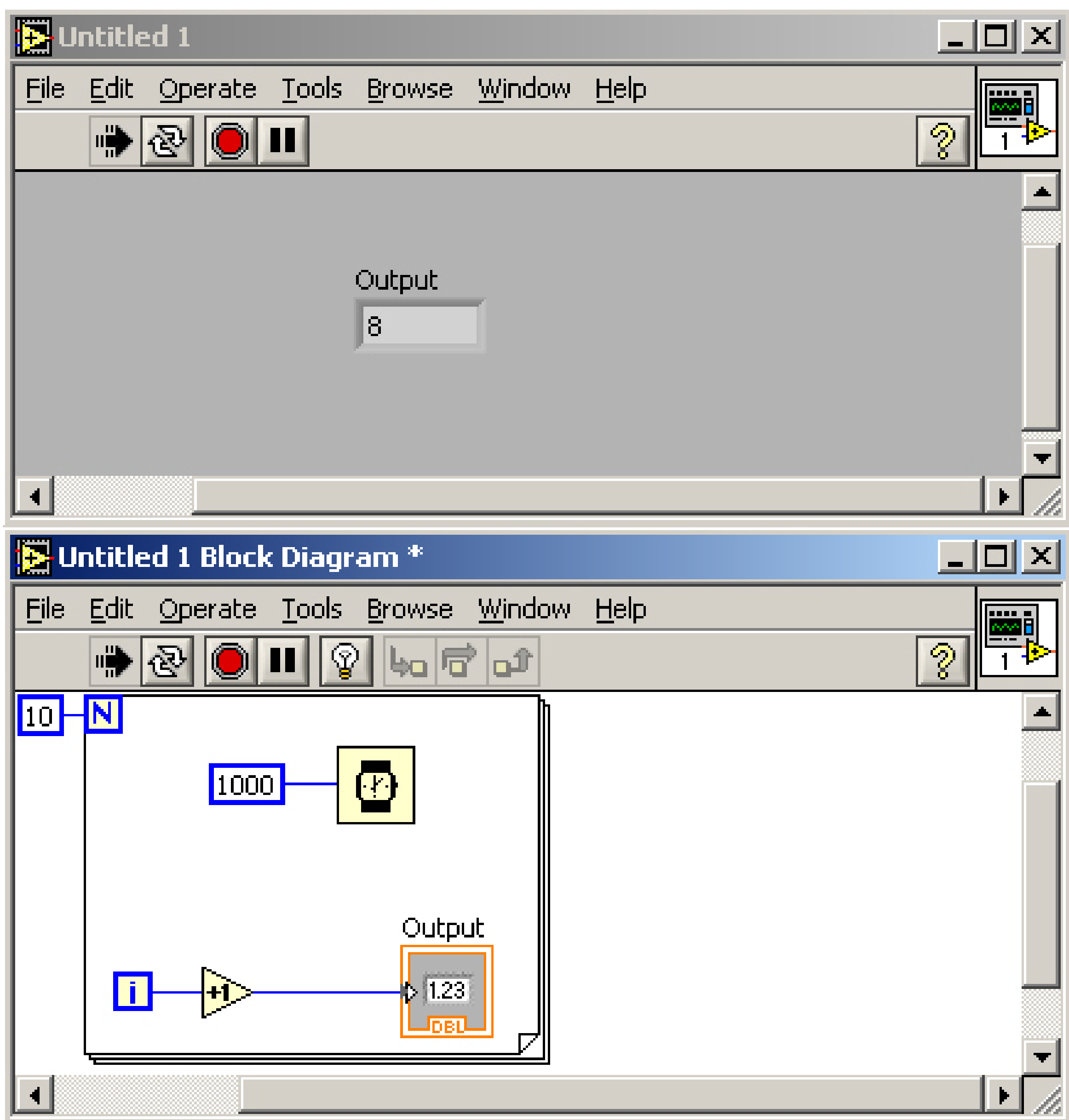
It is not necessary that every initial input be either the output of a control or the output of another subVI. If desired, you may also wire a constant to an input of a subVI. To add a numerical constant, go to the numerical palette and find the numerical constant object (the blue box with 123 in it). Put it on the block diagram. You can then right click on it and go to "Representation" to choose the data type (double precision, long integer, short integer, etc). You can then wire this constant to any numerical input of a subVI. There are other types of constants, such as array constants, enum constants, ring constants, etc., and they all work the same way.

Finally, if you ever get stuck, you should turn on the Context Help feature. To enable this feature, go to the help menu and click "Show Context Help." As long as this mode is on, whenever you hover over a subVI a window will show you help information pertaining to the usage of that particular VI. This is quite useful as a quick reference for times in which you know what a subVI does, but don't know exactly how to use it.

Section 1.3: Structures and Execution Control

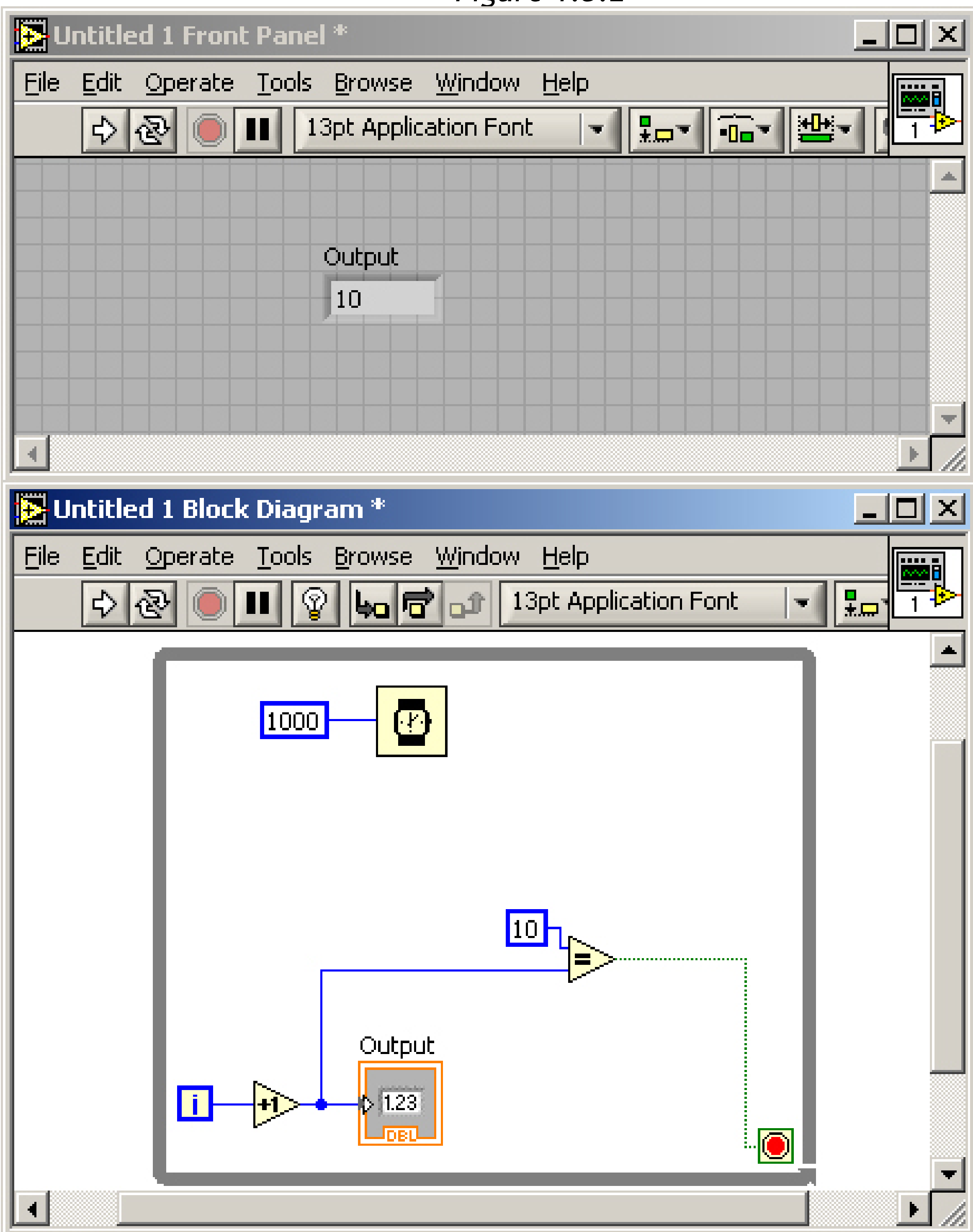
Just as with any other programming language, LabVIEW comes complete with for loops and while loops. A "for loop" loops some code a set number of times, and a "while loop" loops until a certain condition is met. In LabVIEW, loops are represented by a box that surrounds the code that is being looped. For example, the following code counts from 1 to 10 on 1 second intervals:

Figure 1.3.1



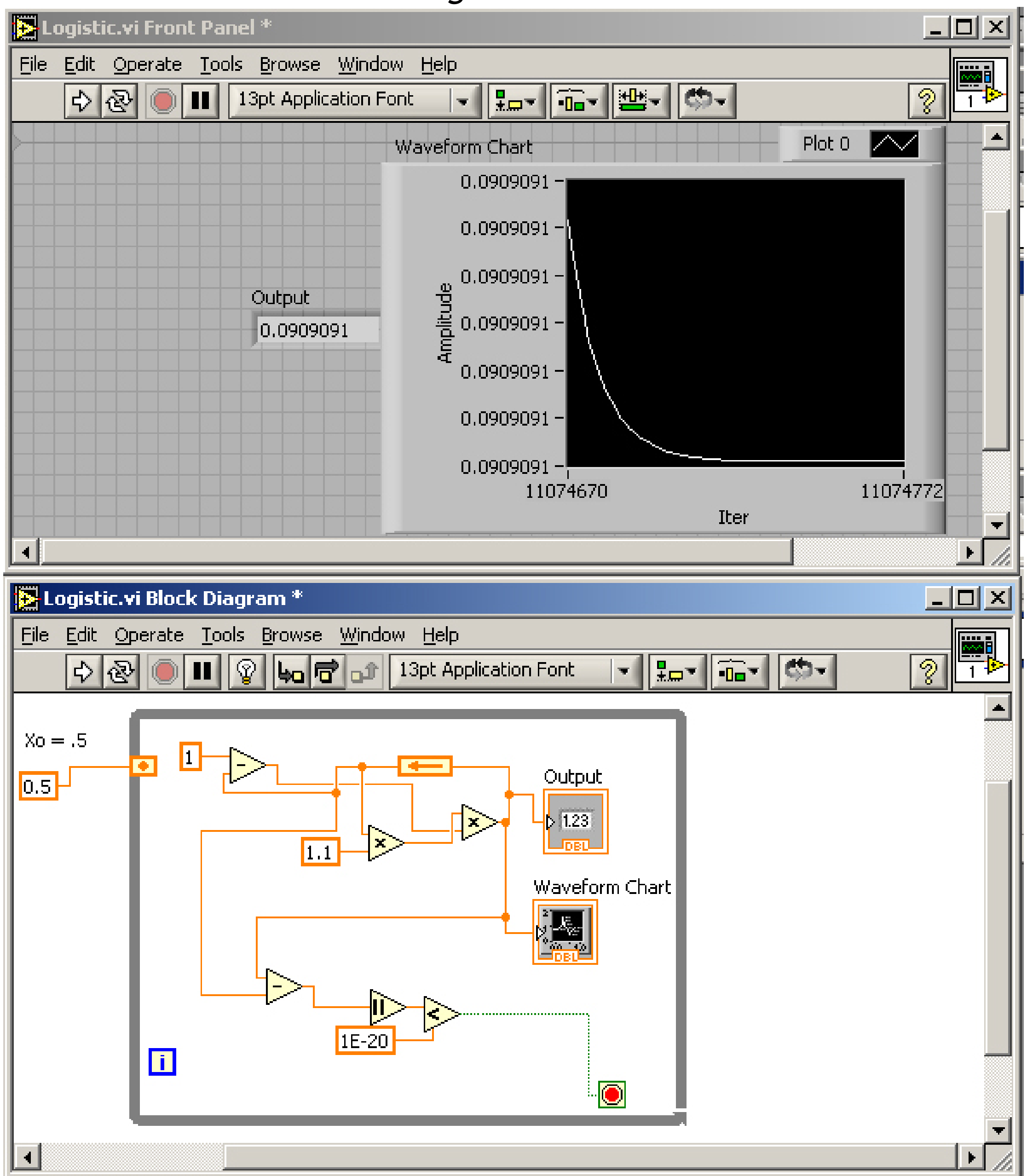
The demonstrated loop is a for loop. N is the number of iterations you would like to do, and i is the zero based index of the current iteration. An equivalent way to formulate this algorithm, using a while loop instead, can be done like so:

Figure 1.3.2



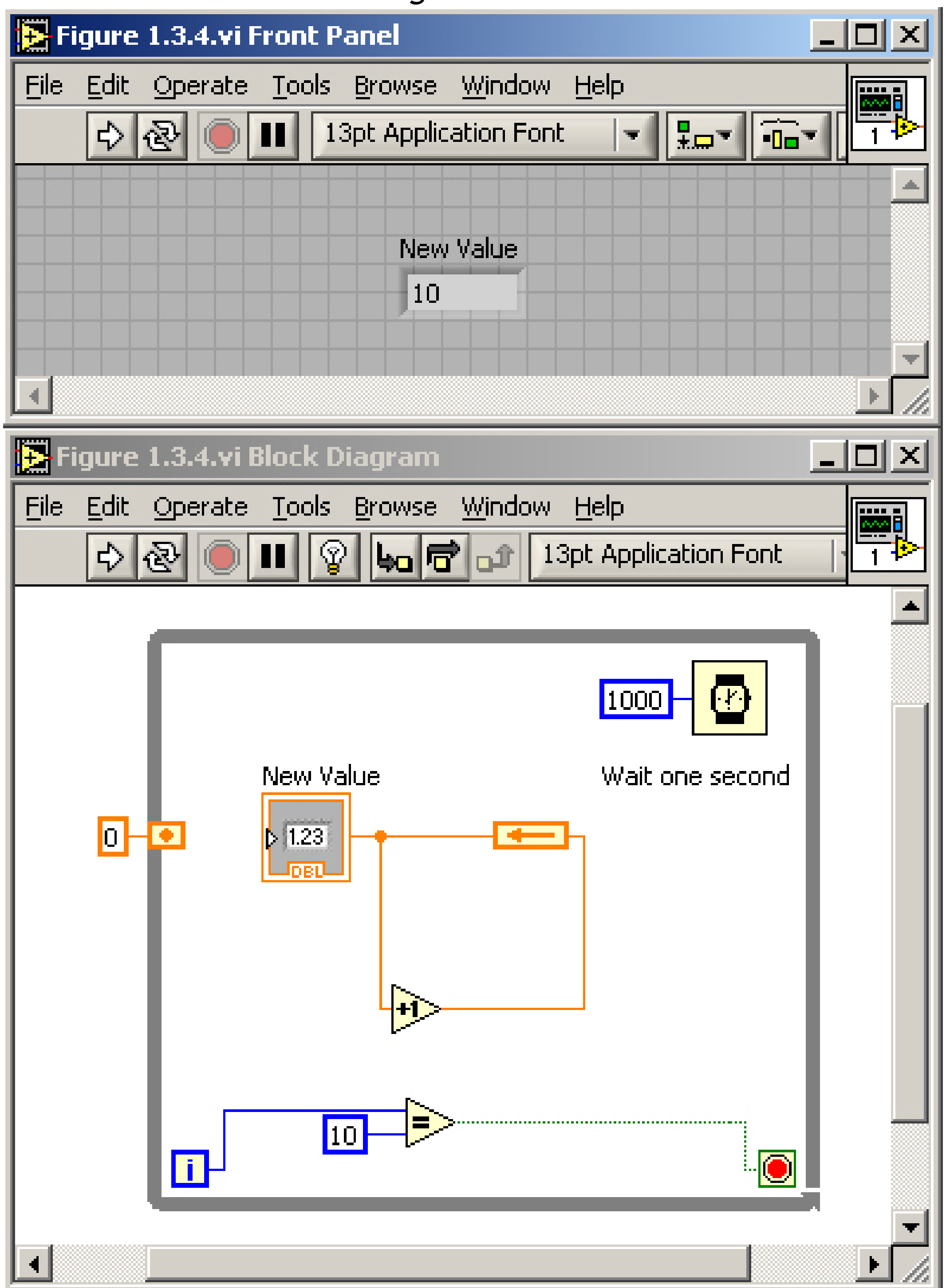
In a while loop, the stop sign is the termination condition. In this case, the stop condition is that the iteration index + 1 = 10, which is equivalent to a for loop with N = 10. By default, the loop terminates if the condition is true. However, you may change this to continue as long as the condition is true by right clicking on the stop sign and selected “continue if true.” This code takes the logistic map, , with alpha = 1.1 and terminates when the difference between and is less than .01. We will return to this example later for a more thorough discussion.

Figure 1.3.3



Now is a good time to introduce feedback nodes/shift registers. These two objects provide a way to pass data from one iteration of a loop to the next. For example, say that you want a loop to add one to a number every iteration, i.e., x0=0, x1=1….xn=n. To do this in LabVIEW, you would first wire a 0 to the initializer terminal (the terminal created on the edge of the left side of the loop that looks like a rectangle with a diamond in it). This dictates the first value that the feedback node will return. You then wire the new value to the “back” of the feedback node (meaning the tail of the arrow as opposed to the head). Thus every iteration of the loop will give a new value to the feedback node, and the feedback node will report that new value when the next iteration begins. Here is an example of using a feedback node.

Figure 1.3.4

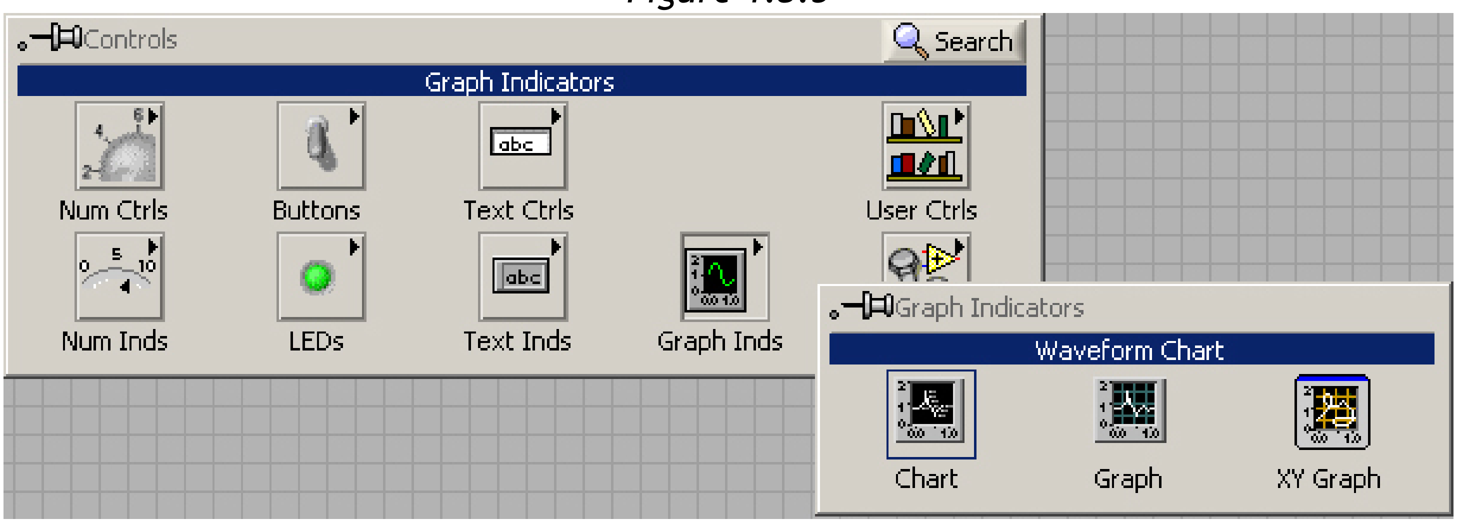


Warning An easy mistake to make when using feedback nodes is to forget to wire the initializer terminal. If you don't wire it, it will pull whatever value it last reported when the program last ran. Also, what it does reports when the program first starts (when it doesn't really have any idea what to report) is somewhat unpredictable in that its dependant upon many things. In general it is best practice to wire the initializer terminal; do it unless you have a specific reason to not do it.

As you may be able to guess from the code, this example does the same exact thing as the example pictured in figure 1.3.1, except it starts at 0 instead of 1.

Now that you know about the feedback node, the last thing that we need to discuss, in order to fully interpret the example demonstrated in Figure 1.3.3, is the Waveform Chart. A Waveform Chart is an indicator that you may place on the front panel by right clicking on the front panel, selecting "Graph Inds," and then selecting "Chart," like so:

Figure 1.3.5



The corresponding indicator icon appears in the block diagram, as you can see above in Figure 1.3.3. Whenever the Waveform Chart is passed data, by default, it scoots over one x value and then plots the new value on the y-axis. This default action can be changed to plot the new value vs. absolute time or relative time as well. You change this behavior by right clicking the chart, selecting properties, selecting the "Format and Precision" tab, and then selecting absolute time or relative time.

Now to introduce the final structure that we will be discussing: the case structure. This structure fulfills the role of both If...Then statements and Case statements in your typical text-based programming languages. Basically, for the If...Then statement functionality, you wire a boolean value to the control terminal of the case structure that controls which part of the case structure will be executed; you only have two options in this case. If you want to have more than one possibility, you can create more than one case, and wire another data type to the control terminal. Here is an example of a simple case structure.

Figure 1.3.6a

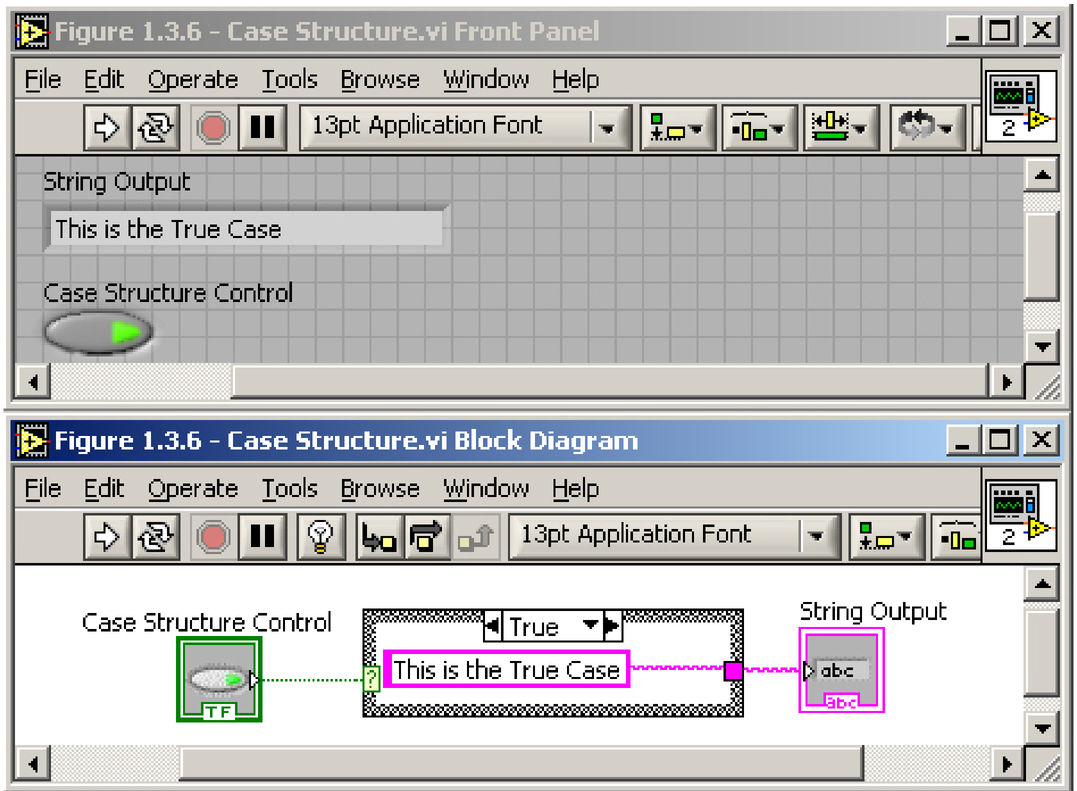


Figure 1.3.6b

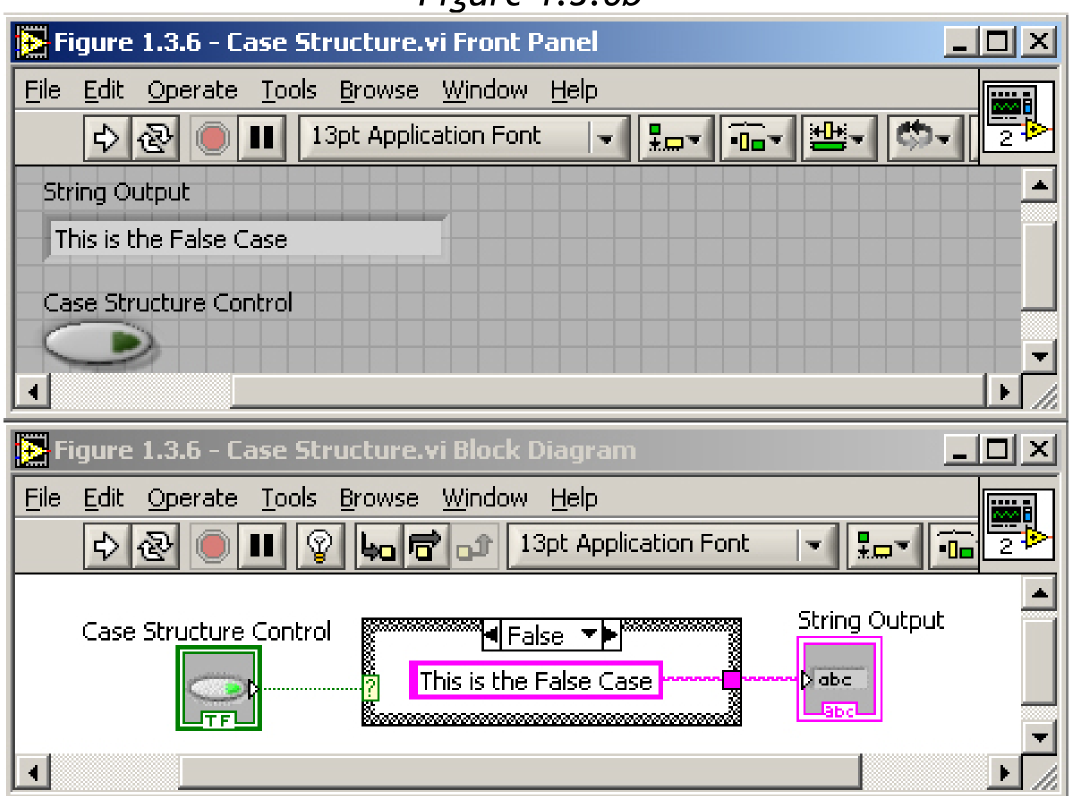
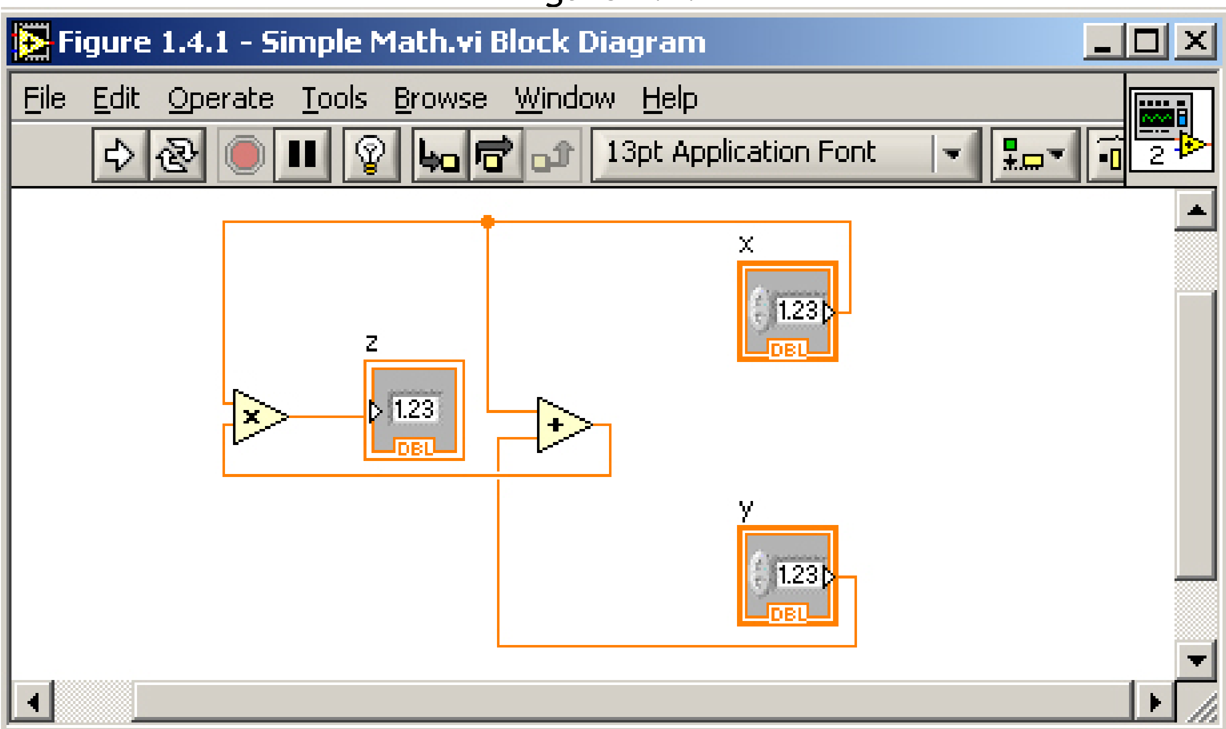


Figure 1.3.6a shows the true case and 1.3.6b shows the false case. As you can see, you put whatever code that you want to execute when the ? box has a true value wired to it in the true case structure box and vice versa for the false case (you change the case by using the arrows at the top of the case structure). You do a similar thing when you want more than one case, but I will leave that up to you to tinker with if it becomes useful to you.

Section 1.4: Data and Execution Flow

The way that LabVIEW handles data flow and order of execution is unique when contrasted with other text-based programming languages. In text-based languages, every command is executed sequentially, and the only way to perform tasks in varying order is to make functions or subroutines out of them. In LabVIEW, the order on the block diagram is irrelevant (although it is somewhat accepted to make programs go from left to right, in general). Take a look at the following example.

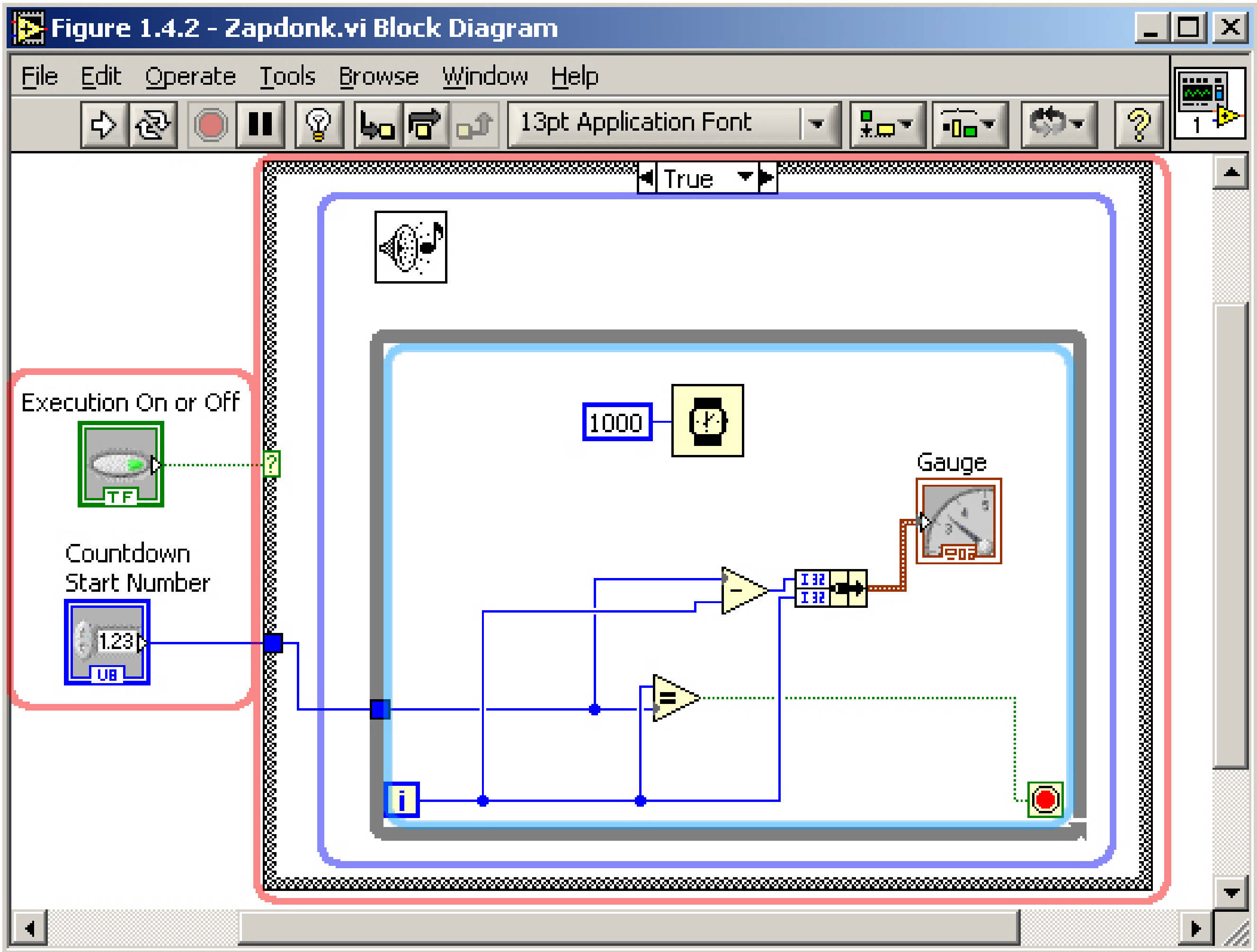
Figure 1.4.1



This program does the same exact thing that the example in Figure 1.2.4 did, except the order is rearranged.

LabVIEW decides execution order based upon the order in which each terminal receives data. It will now be useful to define a term, "zapdonk," to simplify later explanations. This is not a LabVIEW term, but rather one I have coined to better explain this concept. A zapdonk is a collection of commands that has no conditions on its execution at some given part of the execution of the program. This means that once you reach a particular zapdonk in a program, all members of that zapdonk must execute before the program leaves the zapdonk to continue onward to the next zapdonk. As confusing as this is, it deserves an example.

Figure 1.4.2



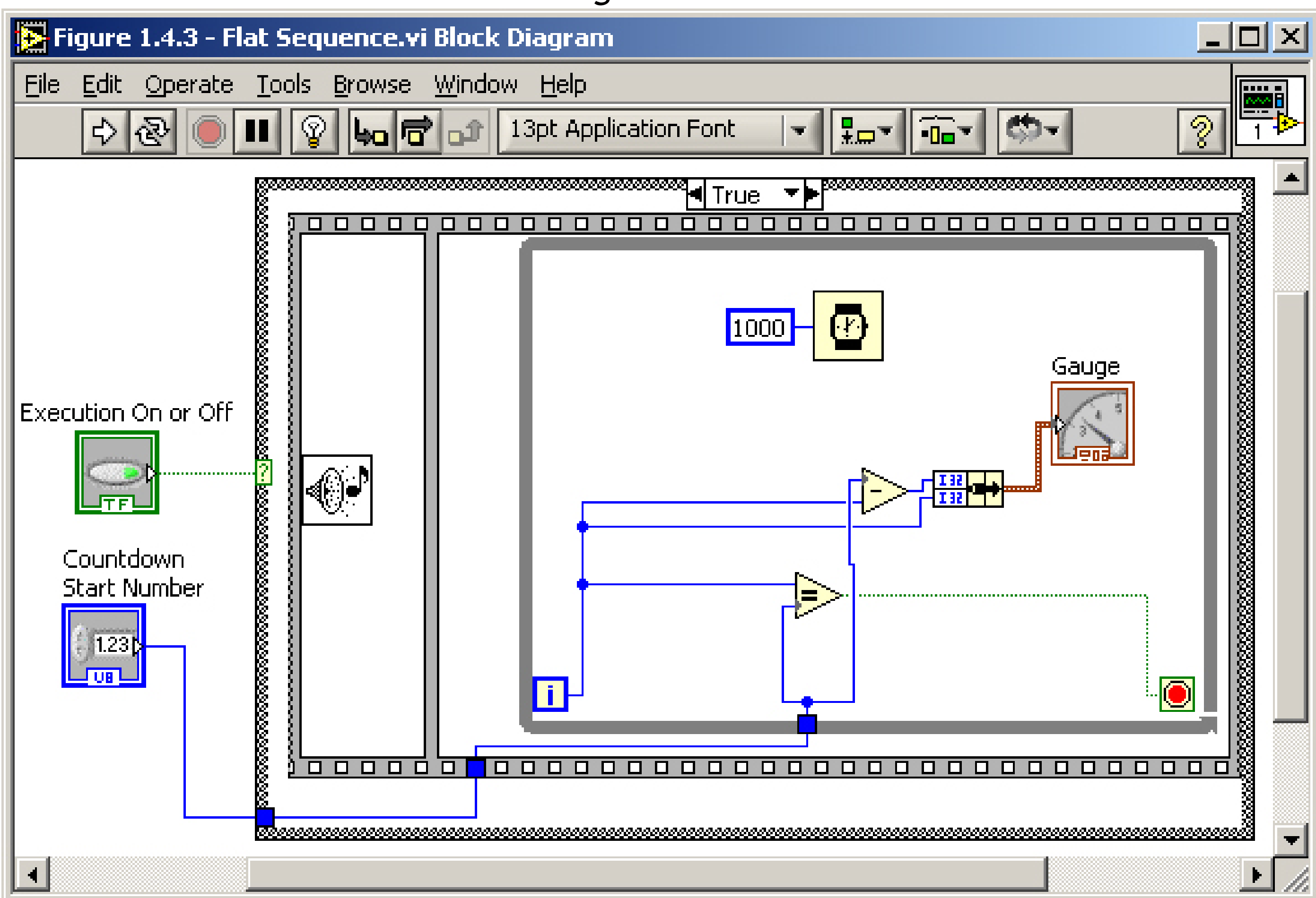
In this figure, each separate zapdonk is surrounded by a separate color. In LabVIEW, each member of an executing zapdonk that can report data immediately does so. Any member that does not yet have each data terminal supplied with data yet waits until every data terminal is supplied with data before it executes. If it is not supplied with data, it will not execute, therefore it crashes. This is a similar crash to trying to reference a variable before it has been assigned a value in a text-based language. Here is a breakdown of what happens in this example.

1) Red Zapdonk executes
2) Boolean and numeric-integer control report their data
3) Case commits to perform the true or false statements
4) If true, blue Zapdonk executes
5) The beep subVI executes at the same time (separate threads) that the while loop executes
6) When the while loop executes, the blue Zapdonk executes
7) The "1000" constant/wait path and the counting path both execute at the same time
8) When the loop finishes, the blue Zapdonk will be finished
9) When the blue Zapdonk finishes, the red Zapdonk will be finished
10) The program terminates

This is much easier to see if you go download the example, turn on highlight execution mode, and watch it execute to see that order that it does things in. The most important thing to remember is that a LabVIEW program executes in the order in which its components are wired, and data is reported "on demand." Considering the fact that all members of a zapdonk execute simultaneously, it can sometimes be a problem to get things to execute in the order that you desire.

To help out with this problem, LabVIEW gives you two tools: the flat sequence and the stacked sequence. Both of these do essentially the same thing, they simply look different on the block diagram. You find both of them under "Structures" on the "All Functions" palette (the same place that for loops and while loops reside).

Figure 1.4.3



This example does the same thing as the example shown in Figure 1.4.2, except it forces the program to execute the beep subVI prior to executing the counting code. A flat sequence executes from the left to the right.

To use a flat sequence structure, simply select it from the Structures area of the functions palette and drag a box somewhere on the block diagram to create the structure. When it is created, simply right click on the border to add frames. After all the frames that you desire are created, add the code to the sequence in the order that you want it to execute, and you're done.

In general, you will want to try to use the wires to control the data flow because it makes your programs much simpler. However, flat sequences are useful in many cases, especially when you need to synchronize different aspects of your program; just make sure you resist the urge to overuse them

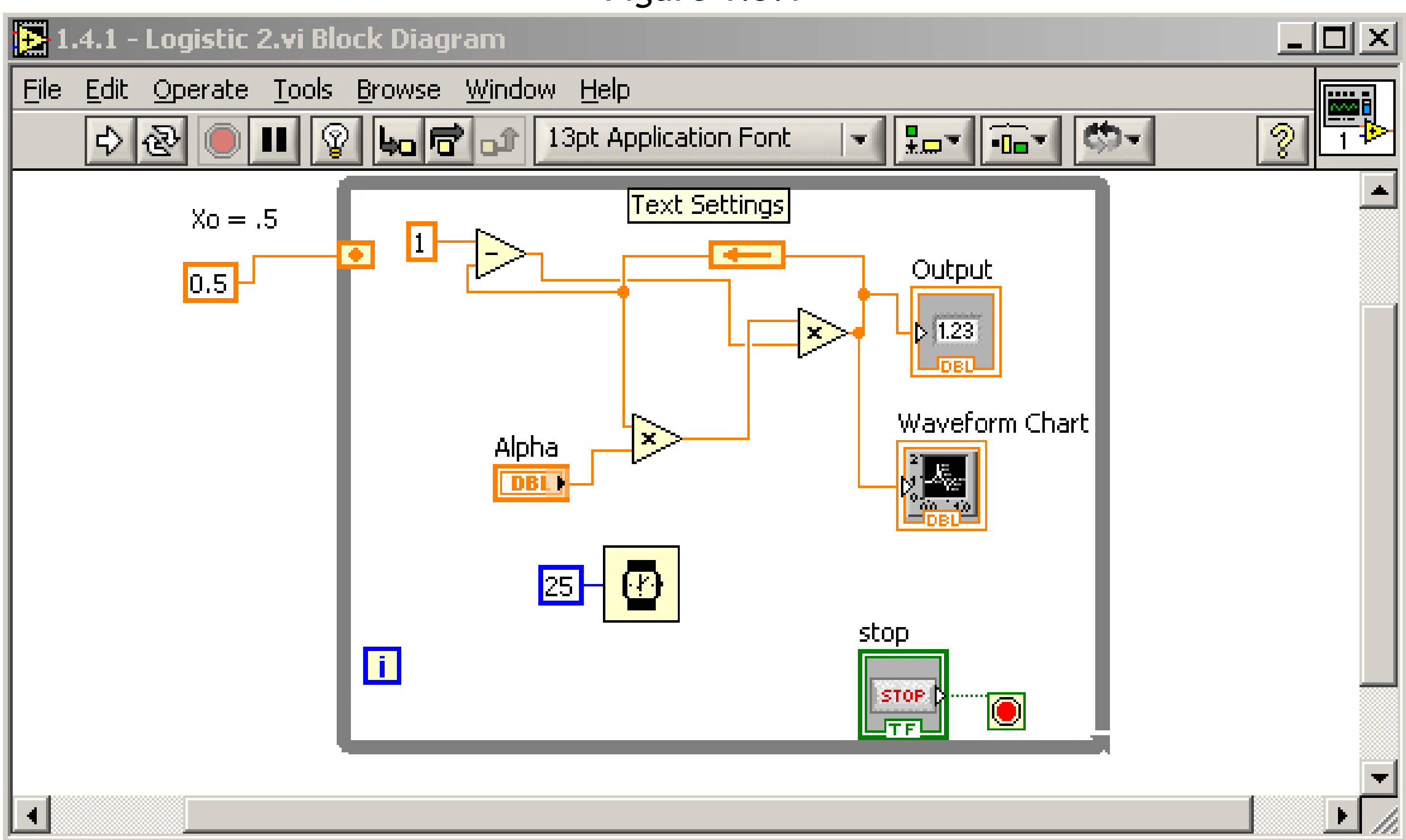
Section 1.5: Interactive LabVIEW Programs

Thus far, the only programs that we have seen have been programs that you simply run, watch it finish, and look at the result. Our goal in this section is to establish a program that will dynamically adapt to user input.

The main component in such a program is, without a doubt, the while loop. In general, you create a while loop around your entire program that has the termination condition wired to some sort of stop button. Let's return to our example regarding the logistic map.

Here is a version of the program that lets you vary "a" as the program runs:

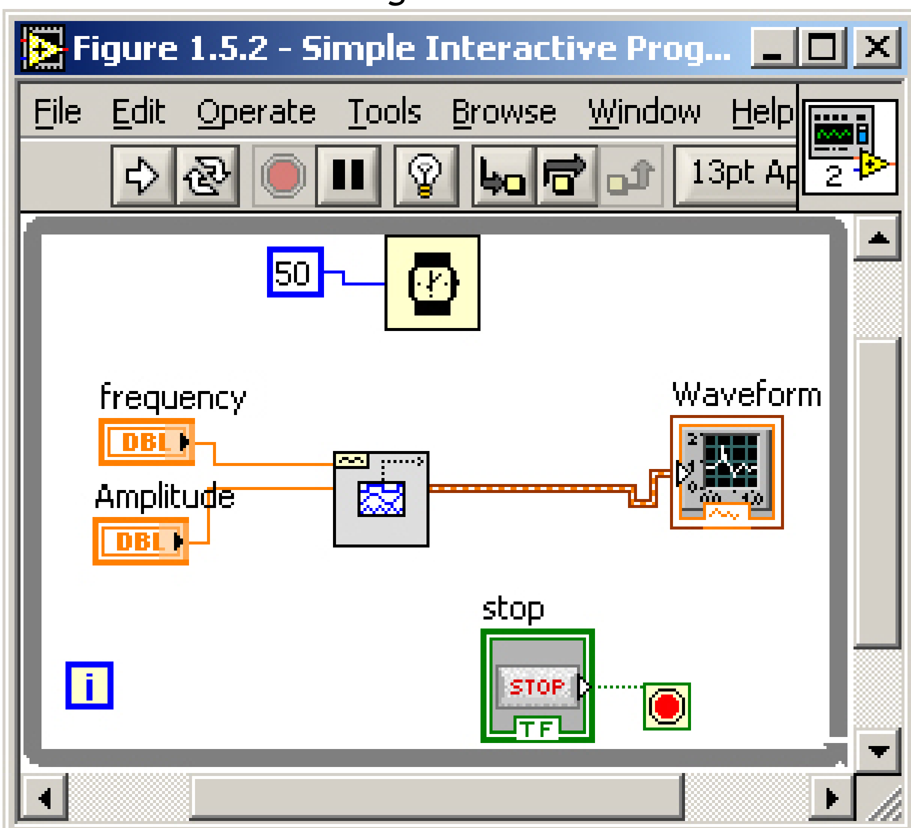
Figure 1.5.1



This program continuously iterates the logistic map and displays the results in real-time on a waveform chart. As you can see from the code, in general, all that you need to do to make a program interactive is assign a control to some parameter and put the entire program in a while loop. This allows the program to continue until a user tells it to stop.

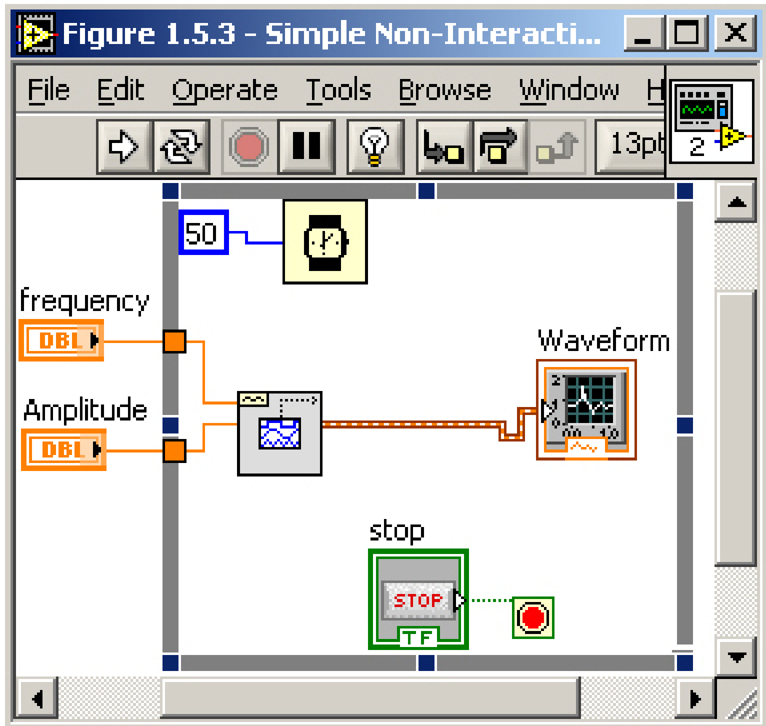
Thus all one needs to do to make a program interactive in LabVIEW is insert a control in whatever loop contains the contains the continuously executing code that you want to control. It is important that you insert the control in the loop and not outside the loop (remember what was said in Section 1.4 about execution order). To demonstrate this, the following example allows for run-time change of parameters.

Figure 1.5.2



And the next example will have the same functionality, except one cannot change the parameters while the program is running.

Figure 1.5.2



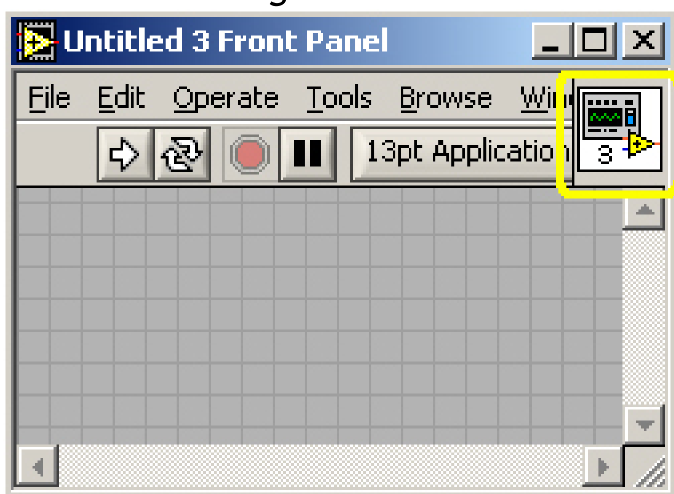
While writing a LabVIEW program, it is generally best to put the control in the same zapdonk as the parameter that it is controlling, in order to provide the user with control over that parameter during the execution of the program. However, there will be many cases in which you will not want the user to control parameters of certain things during execution time (when controlling parameters of a data acquisition device that is actively taking data during the main program execution, for example). Just make sure that you think about exactly what you wish to accomplish with a control when you place it in the block diagram.

Section 1.6: SubVI's

Thus far, we have focused on using the various subVI's that are packaged with LabVIEW. Now we shall learn to make our own. SubVI's are analogous to subroutines or functions in text-based programming languages. There are many reasons to use custom-written subVI's in your programs. For one, you can imagine that as programs get more and more complicated, you tend to run out of room on the block diagram if you put every single bit of code directly in the block diagram you are working on. SubVI's work to organize your block diagram. Secondly, if you work hard to write a good, adaptable subVI you may reuse your code at a later time to do a similar task.

You create a subVI very close to the same way that you create a top-level VI, only you do a little bit extra when you are done working. You begin by creating controls and indicators for every input and output that you want your subVI to have. Then, you use the connector pane to link terminals of your subVI to the corresponding control or indicator. The figure below shows you where the connector pane is located.

Figure 1.6.1



Until you tell LabVIEW otherwise, the connector pane shows the icon for your VI. To show the connector, navigate to the front pane, right click anywhere in the connector pane, and select "show connector." You will see a series of boxes appear in the connector pane. Each of these boxes corresponds to a potential connection that can be made to your subVI. If you right-click somewhere on the connector pane when the connector is showing, you may add/remove terminals and control which connections are required (or optional...called "recommended" in LabVIEW) for execution. Required connections will throw an error when there is nothing wired to it when it is called.

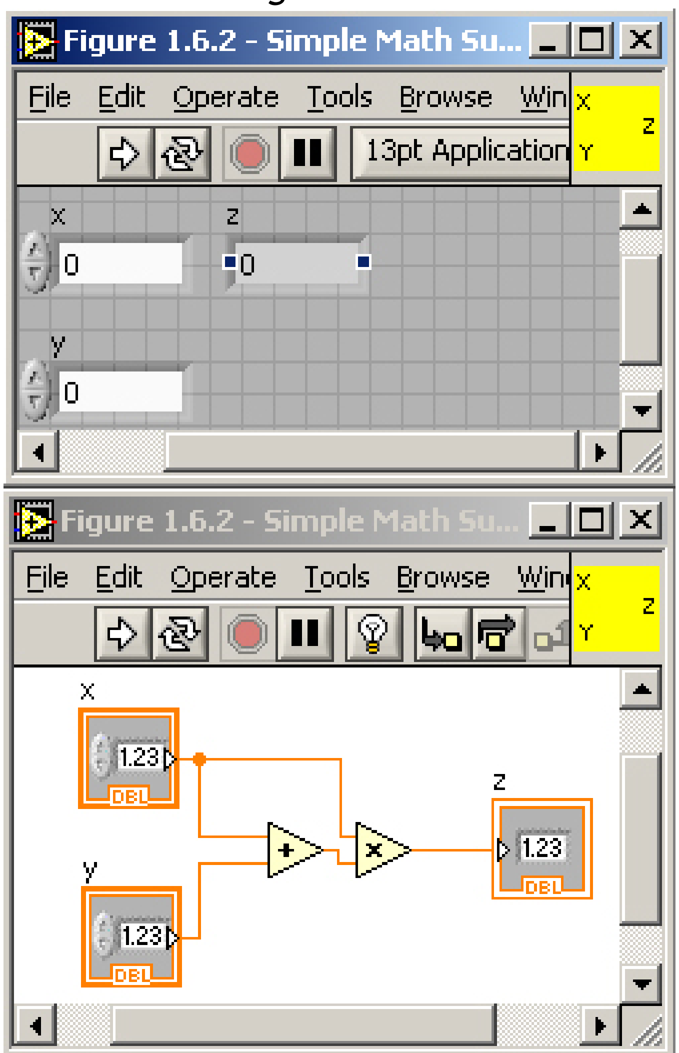
To wire a control or indicator to your connector simply click on a connector box (you'll see the wire spool cursor pop up) and then click on the corresponding control or indicator. You will see the color of the appropriate box in the connector pane change to the color that corresponds to the data type of the control or indicator that you wired to the connector.

Now to design the icon. It is generally best to design an icon in a separate graphics editor such as Photoshop or GIMP. All you must do is design a 32 x 32 pixel graphic that you want to use as the icon and then drag it from anywhere on your file system to the connector pane. However, if you do not have such software available to you, LabVIEW provides a very rudimentary and not very easy to use icon editor.

To access the LabVIEW editor, all you must do is right-click in the connector pane and select “edit Icon.” The icon editor is pretty obvious in its frustrating usage, so I’ll just let you play around with using it.

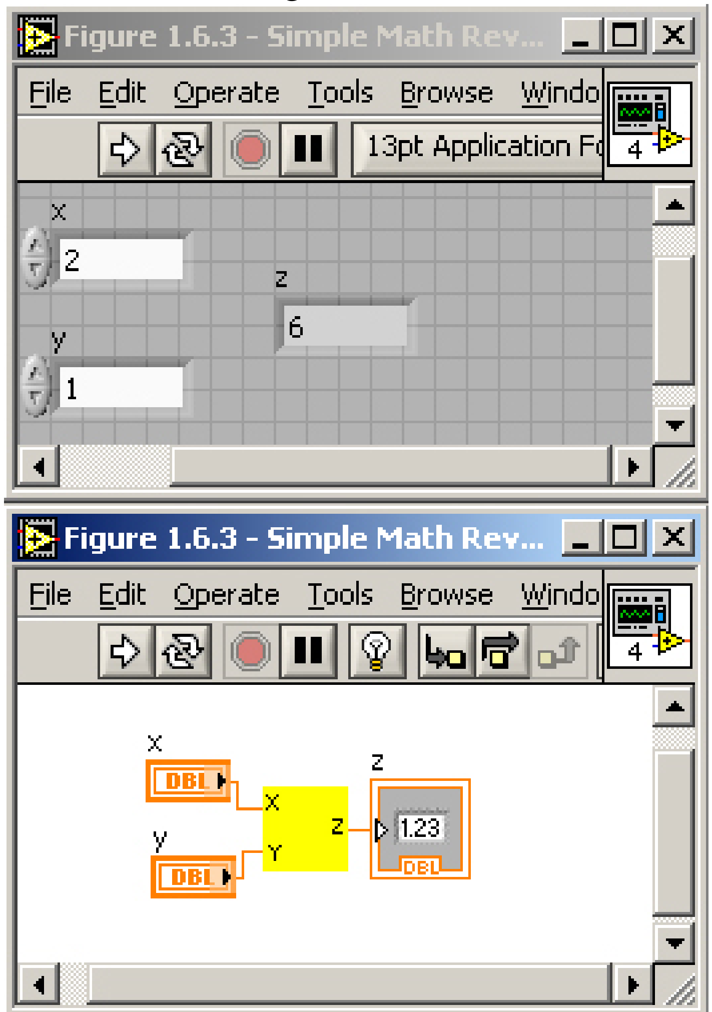
You now know all the basics for creating a subVI; here is an example. I’ll take the first complete subVI that we made and turn it into a subVI (see Figure 1.2.4).

Figure 1.6.2



As you can see, the only differences between this program and the program of Figure 1.2.4 are that is has an icon that I made and that, although you can’t see it, I connected the X, Y, and Z terminals to the appropriate connector in the connector pane. To place a subVI that you have made on the block diagram, you may do one of two things. You may find it in the file system and click and drag it onto the block diagram, or you may go the “All Functions” area of the functions palette, find “Select a VI...” and then navigate to the location of your subVI that you desire. Either way the effect is the same; you end up with your subVI on the block diagram. You may choose which method you prefer. Now here is an example that uses this new subVI from Figure 1.6.2.

Figure 1.6.3



To edit a subVI that you have created when it is already in your block diagram, simply double-click it to open up a new LabVIEW window with the content of the subVI present to be edited. Any changes that you make to the subVI will be reflected in any other VI's that call it, so be careful when you do this.

If you run this program, you will see that it has the same output as the program in Figure 1.2.4; it is simply a little bit neater. While making a subVI to do something this simple is overkill, it is easy to imagine times that the content of a subVI would not be quite so trivial (just double click the Function Generator subVI that we used in Figure 1.5.2 for a prime example of a non-trivial subVI).

There is one other way to create subVI's that is useful when you have already written a program, and you would like to clean it up a bit. To create a subVI this new way, you simply highlight (marquee around) all the code that you have written that you wish to turn into a subVI and then go to Tools > Create SubVI. To demonstrate this new functionality, I will return to the VI found in Figure 1.5.2. All that I will do is select the "50" constant, the Wait subVI, and the function generator then navigate to Edit > Create SubVI. What results is a subVI that contains all the code that I selected; LabVIEW even wires all the terminals to the proper terminals for you.

Figure 1.6.4

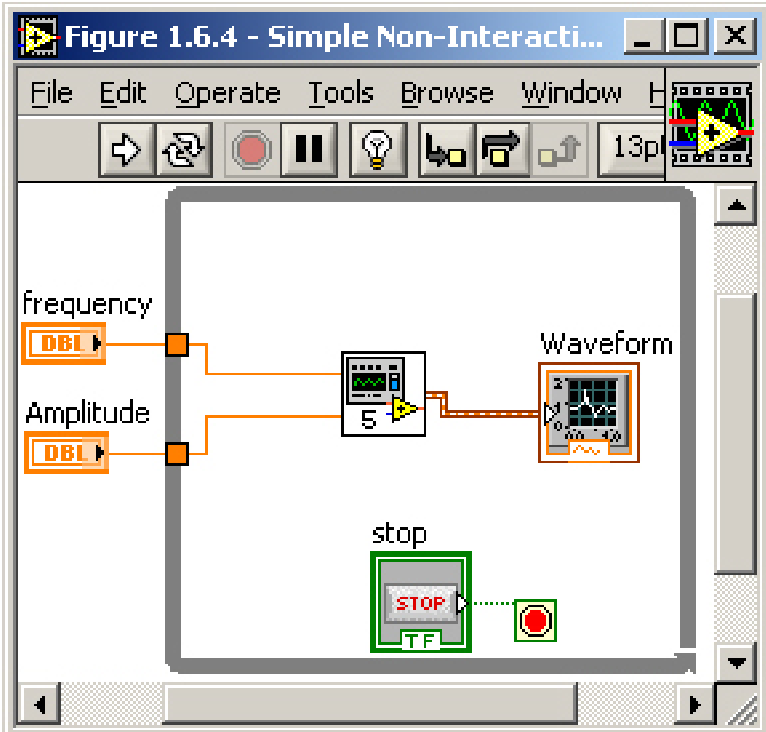
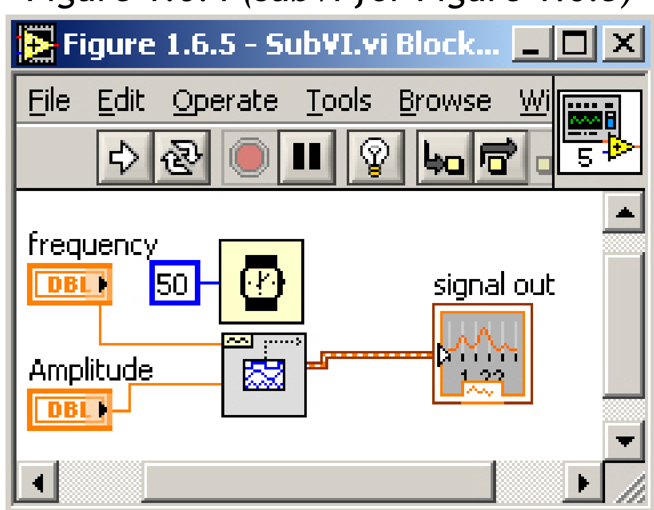


Figure 1.6.4 (subVI for Figure 1.6.5)



As you can see, the subVI simply takes the place of the previous code. The default icon is created for your new subVI; you may double-click on it and edit the icon to be whatever you would like, just like any other subVI.

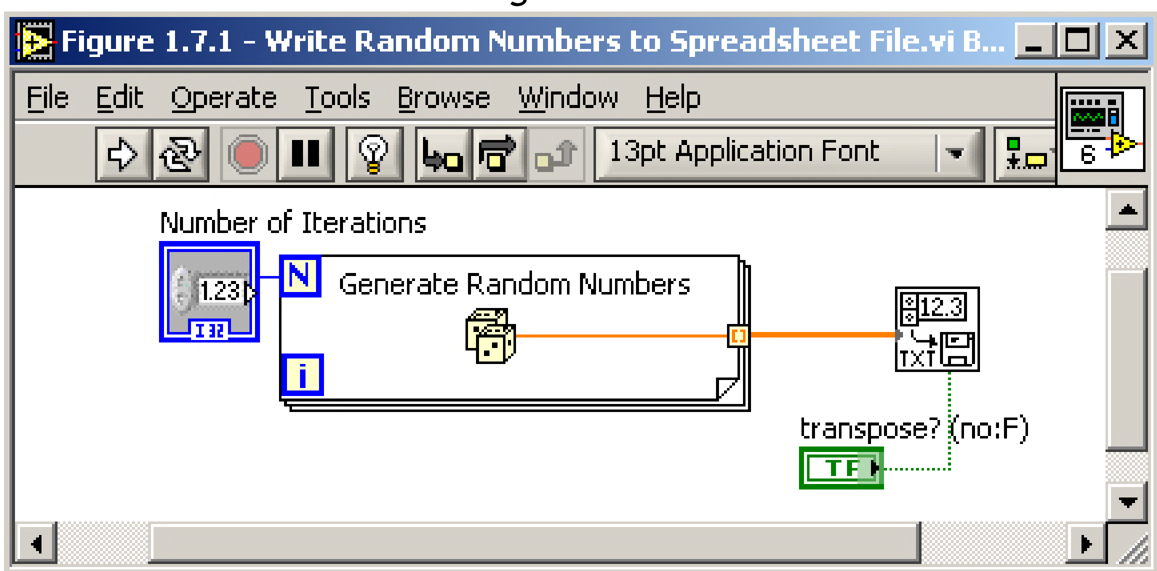
Just remember that when you make a subVI, you remove any potential interactivity with the contents of that subVI--you simply pass it arguments, and it gives you results. Thus you don't generally want to put one of your loops that are designed to provide interactivity (like we discussed in section 1.5) in a subVI, since that removes any control that the user may have over the contents of that loop at runtime.

Section 1.7: Saving Data

There are many ways to save the data that you take in LabVIEW. The primary way that we will focus upon is exporting the data in a spreadsheet format. Luckily, LabVIEW comes with a built-in subVI that handles exporting data in various delimited formats, including comma separated and tab separated (the two that we will be dealing with most often).

Most of the File I/O operations may be found in the File I/O area of the All Functions palette. The main VI that we will be using is the "Write To Spreadsheet File" VI. This VI writes a 1 dimensional or 2 dimensional array to a spreadsheet formatted file using whatever delimiter that you tell it to use. By default, the Write to Spreadsheet File VI uses a tab as its delimiter which is exactly what we want it to use. First, to get a basic understanding of what the VI does, I suggest turning on context help and reading about what it does. When you have a general understanding of how it works, look at the following example.

Figure 1.7.1



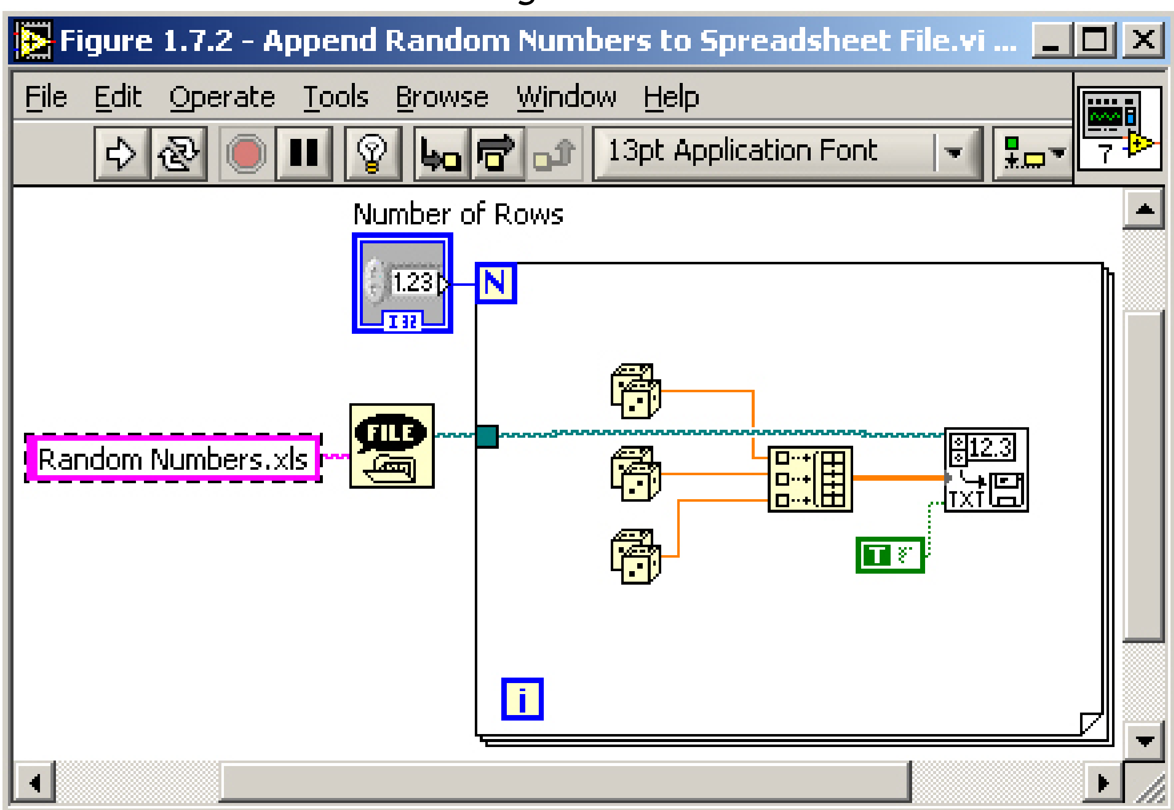
This VI generates N random numbers (the user can control N) and then writes all of them, in the order that they were generated, to a spreadsheet file that is named via a dialog box. The first thing to explain in this example is the indexing role of the tunnel (a tunnel is the little box on the edge of the for loop that gives you access to the data inside of it). Whenever you have indexing turned on (toggled on or off by right clicking on the tunnel and selecting "enable indexing" or "disable indexing"), every iteration of the loop writes appends a new value to an array that grows as long as the loop executes. The net effect of indexing an output is that the 0th element of the output array is the first value that the loop reported, the 1st element is the second value that the loop reported, etc. If you disable indexing, only the final value that is reported as the loop terminates is output through the tunnel.

In this example, we wired the indexed output of the random number generator to the 1D input of the Write to Spreadsheet File VI. This causes the Write to Spreadsheet File VI to write that entire array horizontally if the transpose flag is false or vertically if the transpose flag is true. It is noteworthy that this particular VI only writes numeric data--no strings, waveforms, etc are allowed.

The method described in the previous example is a good way to write data in some cases, but it is generally not the best way to write data that has been acquired in a lab setting. The reason for this is that it doesn't include any sort of disaster recovery, meaning that if for some reason the for loop crashed, all the data that was taken in previous iterations is lost. This is not a big problem for our little random number generation, but it is a huge problem if you have been taking data for a day and a piece of equipment gives you an error that you weren't prepared for that crashes your program (this has happened to me 3-4 times before I changed my way of doing things). Thus we want a way that exports data continuously, not as one big bundle at the very end of execution. We can do this using the append functionality of the Write to Spreadsheet File VI.

Take a look at the following example:

Figure 1.7.2



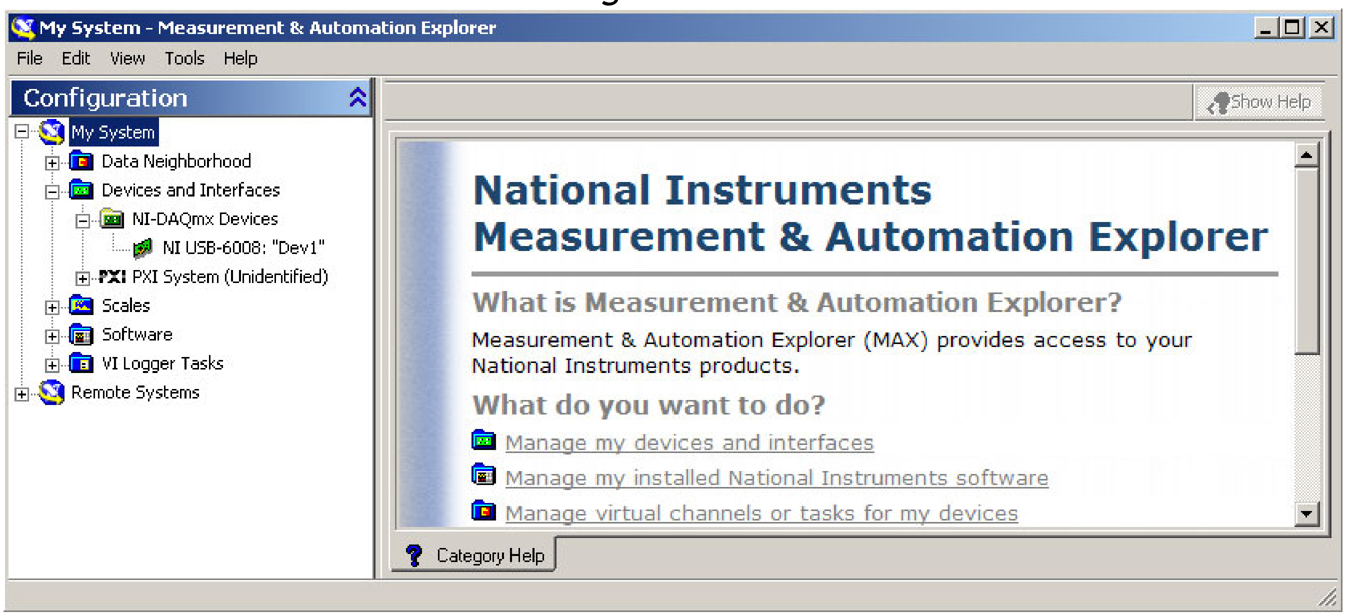
This example gets a filename/path from the user when the VI is first executed. This filename/path is then passed to a tunnel on the loop. Thus every iteration, 3 random numbers are generated (which, since transpose is not on, means that one row with three columns of data is generated) and appended to the file that is specified by the filename that was acquired at execution. On the first iteration, when the file doesn't exist, the file is created.

The major concern with this method is finding some way to make sure that the user does not append data to a file that already exists, unless they specifically mean to. In most cases, you will want to include functionality that keeps users from making this sort of mistake by throwing an error if the file already exists when you first procure the path and filename. Unfortunately, this is a relatively complicated process that is out of the scope of this particular paper; ask me how if you would like to know, and I'd be happy to tell you.

Section 1.8: Data Acquisition

One of the strengths of LabVIEW lies in its seamless integration of data acquisition. The process is simplified even more if you use National Instruments hardware, since they have absolutely superb drivers written for all of their equipment. When you are using NI hardware, you may use a handy VI called the DAQ Assistant, which is found under the "NI Measurements" area of the All Functions Palette. This VI allows you to create virtual channels that serve as a handy, migratable way of handling data acquisition. First, however, I must introduce you to MAX (Measurement Automation eXporer). Here is a screen shot of MAX that also shows where the USB-6008 appears.

Figure 1.8.1



If you right click on a device, you will see a selection entitled "Test Panels." The test panel allows you to do simple read/write operations for testing purposes. You may also rename a particular device from the default of "Dev1" to something more user-friendly. This device name is what you will refer to when you are using LabVIEW to interface with the device.

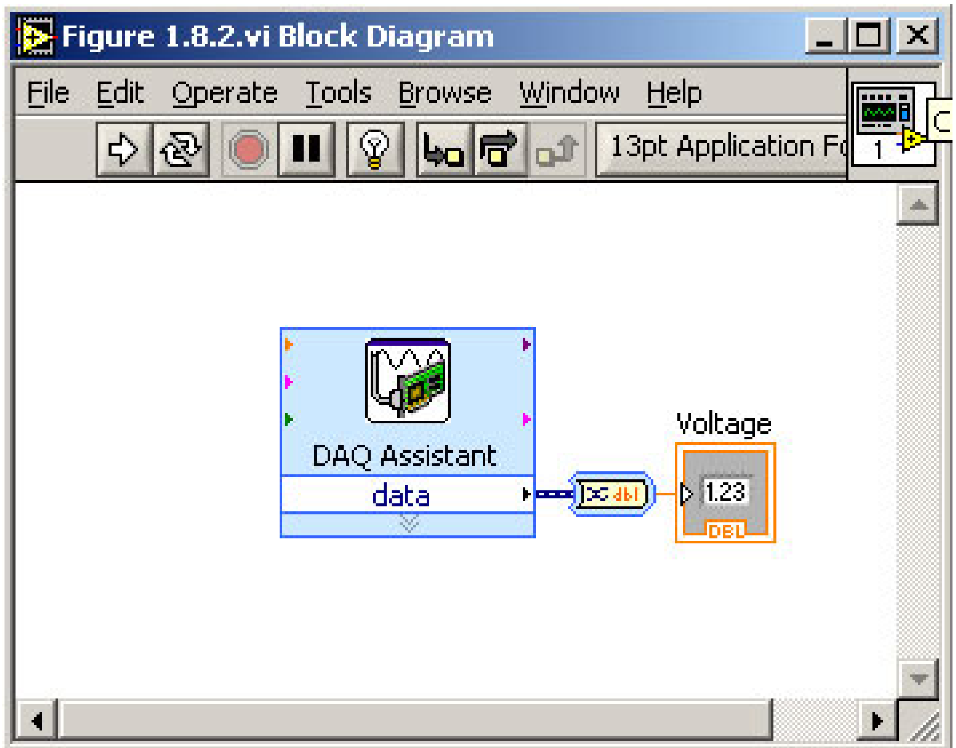
Getting data from a NI data acquisition device is fairly simple. The first step is to put a DAQ Assistant VI on your block diagram. You may find this VI under the "All Functions" in the "NI Measurements" section. When you first place the VI on the block diagram, it pops up a dialog box that lets you configure your data acquisition job. Here you choose what you want to measure (we will be using voltage analog input), what channels you wish to use, what acquisition mode you wish to use (continuous, N samples, etc). After you finish, you may reconfigure the VI by double clicking the DAQ Assistant VI.

The DAQ Assistant VI outputs data in a data type called "Dynamic Data." This data type is basically a way of keeping all the channel information, number of samples, acquired data, device information, etc., all in one place. Often times, you must convert this data type to other data types to make use of the data. For example, if you take only one sample and you just want a number to represent the measurement (commonly a voltage) you must extract the sample value and convert that to a scalar. You use the "Convert From Dynamic Data" VI to do that.

The "Convert From Dynamic Data" VI is found in the "Signal Manipulation" area of the default Functions popup, next to the "All Functions" area (in other words, it is in the first area that pops up when you right click on the block diagram). Once you place it on the block diagram, it will pop up a dialog box that allows you to choose what output format you wish to use. Generally, the simples and most common use is the "Single Scalar" output format, but you may choose whatever is most useful to you. After you click "Ok," all you must do is wire the dynamic data source to the VI, and then wire the output to whatever you wish.

Thus, in general, there are two main steps to performing data acquisition: configure the DAQ Assistant and convert the dynamic data to a more specific format. Here is an example that does just that; as you can see, it takes very little code to perform a data acquisition task.

Figure 1.8.2



This VI Simply reads channel zero (as configured by the dialog box associated with the DAQ Assistant) and outputs its data to an indicator. When you are using NI-DAQmx-ready hardware (such as the USB-6008 that we are using), it is as simple as telling the DAQ Assistant what you want to do and then letting it do the rest.

Section 1.9: Closing Comments

Now you know the most important basics of programming in LabVIEW. There are many, many aspects of LabVIEW that I have not covered here, but I believe that after you have read this, you will understand enough to be able to go out on your own, read some documentation, and understand what you read.

When you finally get to the lab exercise, you will undoubtedly run into something that you need to do that you do not know how to do. I encourage you to first familiarize yourself with the help menu (there are many nice examples, as well as very thorough documentation available to you) and then with my e-mail address, mhollin3@utk.edu. I will be happy to help in any way possible.

Furthermore, if you want more information, there are some very good forums available to LabVIEW users to be found at http://zone.ni.com/devzone/cda/main. There are many people there who know what they are talking about, and chances are, whatever your problem is, it has already been solved; the problem lies in tracking down the answer to it.

Finally, over time I will have more and more LabVIEW content available on my web site, http://www.evanescenthorizons.com/. For now, I have all the examples contained in this text available online under Labs > Introduction to Modern Data Acquisition. That is also the location of the electronic version of this text, assuming at the moment that you are reading a printed version...otherwise, I guess you found it :).

MATLAB

Section 2.1: Introduction to MATLAB

MATLAB is an industry standard numerical computing tool that is useful in all kinds of scenarios. Its prime use that we will be focusing upon is that of data analysis. MATLAB comes with many built in functions that can import data formatted in common spreadsheet formats (such as tab and comma separated), and we will be using these functions to import data for analysis when we finally come to the lab exercise itself. In the meantime, I will work to familiarize you with the basics of the MATLAB platform.

While MATLAB contains very high quality, predictable functions, a large percentage of its strength lies in the sheer number of content that it contains. There are thousands of functions, including ODE/PDE solvers, three dimensional interpolators, data fitting algorithms, numerical and symbolic differentiation/integration, etc. Plus, it comes with a complete and extensive visualization functionality that includes both 3d and 2d graphing capabilities. It is this graphing functionality that will be the main focus of our dealings with MATLAB when it comes time to work on the actual lab exercise.

In addition to performing singular, one shot calculations, one may program MATLAB to do certain tasks or sets of calculations that you wish to use again in the future. “Programming” in MATLAB is more akin to scripting than anything else. In fact, code written in MATLAB is commonly referred to as a MATLAB script or m-file (named for the fact that the typical MATLAB script file extension is .m). Scripting in MATLAB is straightforward once you get used to using the command window, since you simply type the same thing in the script that you would type in the command window. Since you will be introduced to MATLAB first by using the command window, scripting, for the most part, will come along of its own accord.

My example scripts will be in the following format:

Example 2.x.x

```
%Filename
This is some sample MATLAB code
This is some more MATLAB code
Yep you guessed it...more MATLAB code
```

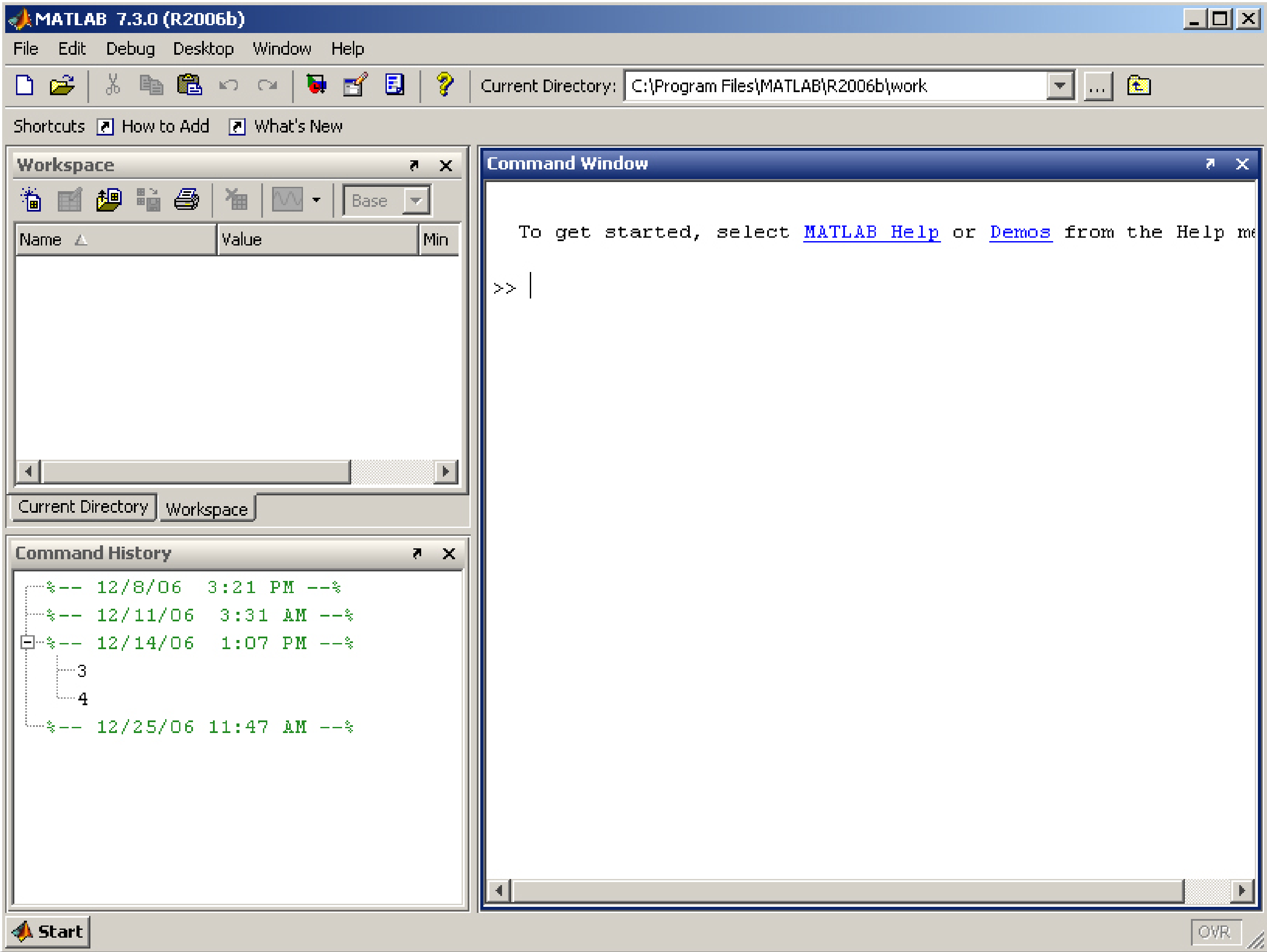
My example code, on the other hand, will simply be in line, like so:

```
>> MATLAB command
|| Output for MATLAB command
>> Second MATLAB command
|| Output for Second MATLAB command
```

In the example code, a return is implied for every new line, in other words, you would type “MATLAB command” then type enter before going on to the second line of commands. The >> denotes input while the || denotes output.

Now that all the introductions are out of the way, I will begin by showing you around the MATLAB interface. When you start up MATLAB, you should see something similar to the following:

Figure 2.1.1



The biggest window, called the "Command Window," is where you do most of your work. It is here that you call functions, do arithmetic, assign variables, call your scripts, etc. The "Workspace" window shows any variables that you have created. It allows you to see what kind of variable it is, check its value, delete it, or edit it. You may also import variables by loading previously saved workspaces (saved as .mat files). The "Current Directory" tab that is underneath the "Workspace" tab is the area that allows you to see the contents of your current working directory. It also lets you change your working directory to another directory if you would like. Your current directory dictates what scripts will be executed when you type their name into the command window. For example if you have a script called "MyScript.m" in your current working directory, you could type `MyScript` into the command window to execute your script.

You may customize your interface by dragging each of these windows around your work area. Simply click on the window title and drag it around to reposition or tab it in another window. If screen real estate is important to you, you can tab all windows together in one area. You may also click the little arrow in the top right-hand corner of a window in order to give that window its own frame in Windows. Additionally, there are more windows that you may find useful; you may find these under the "Desktop" menu.

Section 2.2: Simple Math with MATLAB

Now to do some math in MATLAB. Take the following input to MATLAB for example:

```
>> 2+2
|| ans =
||        4
```

(remember the the || is simply my way of telling you that the text is output... MATLAB does not really output that)

When you input `2+2` into MATLAB, it returns the answer by assigning the answer to a variable called ans. If you look under your workspace window, you will see that a variable named ans has indeed been created. MATLAB always assigns its results to a variable. It can be a variable that you assign, or if you don't explicitly assign a variable, it assigns it to the `ans` variable.

A variable assignment is pretty straightforward in MATLAB. All you have to do is state the variable name, type an equal sign, and then type what you want that variable's assignment to be. Here are some examples

```
>> x = 2
|| x =
||        2
>> y = 2+2
|| y  =
||        4
>> z = x+y
|| z =
          6
```

Make sure you don't accidentally put the variable **after** the equal sign, ie, `2=x`, as that will give an error.

All of these variables are of a numeric data type; however, there are other types of variables. We will discuss others when the time calls for them. Regardless, all variable assignments are made in the same way, i.e., `<variable name> = <desired expression>`.

Also, if you wish to suppress the output of anything in MATLAB, simply put a semi-colon after the expression that you wish to silence. For example, if you would like to set x equal to 4, but you don't want MATLAB to repeat the assignment to you, do the following:

```
>> x = 4;
```

You will see no output when you do this; however, you may see that the variable was properly assigned by looking in your workspace.

Now, onward to functions. MATLAB uses the common and familiar parenthetical method of passing arguments, which is `function_name(argument_1, argument_2, ... argument_n)`. To use a function in MATLAB, you simply type the function's name and arguments (separated by commas) and hit return. For example:

```
>> x = peaks(30);
>> y = sin(pi);
>> z = yourownfunction(2);
```

MATLAB comes with thousands of very useful functions (which is the most powerful part of MATLAB), so we will be using functions quite extensively.

Now that we have the basics out of the way, we will concern ourselves with the next most important syntax consideration in MATLAB: matrices.

Section 2.3: Matrices and Vectors

MATLAB sees all numeric variables as matrices; single numbers are seen as 1 x 1 matrices. A vector, to MATLAB is any 1 x n or n x 1 matrix. There are many ways to go about creating matrices. One of the easiest ways to deal with two dimensional ones is to use the bracket syntax. It looks something like this:

```
>> x = [1,2,3;4,5,6;7,8,9]
||        1      2      3
||        4      5      6
||        7      8      9
```

You start in the upper left-hand corner of the matrix you are creating, and work your way across, left to right. Commas separate members of the same row, while the semi-colon separates the rows themselves. Alternatively, you may assign elements in an array individually by using a parenthetical syntax in the form of `variable(row,column)`, like this:

```
>> x(1,1) = 2
|| x =
||
||     2
>> x(1,2) = 3
|| x =
||
||     2     3
>> x(2,1) = 4
|| x =
||
||     2     3
||     4     0
>>x(2,2) = 5
|| x =
||
||     2     3
||     4     5
```

MATLAB displays zeroes when it doesn't have any data for an element that must be displayed, such as in the third example above. To make it a little more obvious, here is another example:

```
>> x(6,6) = 2; x(3,3) = 1
||
|| x =
||
||     0     0     0     0     0     0
||     0     0     0     0     0     0
||     0     0     1     0     0     0
||     0     0     0     0     0     0
||     0     0     0     0     0     0
||     0     0     0     0     0     2
```

(as you may have already figured out from the last example, you may use semi-colons as separators for commands if you wish to type multiple command on one line)

You may also use this syntax to create nth dimensional matrices by the generalized version of this syntax: `variable(dimension1, dimension2, dimension3, ... dimensionN)`. Here is an example:

```
>> y(2,2,2) = 2
|| y(:,:,1) =
||
||      0     0
||      0     0
||
|| y(:,:,2) =
||
||      0     0
||      0     2
```

When you tell a variable to be a multi-dimensional matrix in MATLAB, and you ask MATLAB to display the contents of said matrix, it will display a series of two dimensional matrices that collectively reflect the contents of the variable . You can see this in the above example. So that you fully understand what you are seeing, I will explain what the colons mean.

You may use colons to tell MATLAB to display all of the contents of that matrix dimension. In other words, if you want to display all the rows in one specific column, you type `variablename(:,column_index)`. Similarly, if you want to display all the columns in one row, you type `variablename(row_index,:)`. Here is an example:

```
>> x = [1,2;3,4]
|| x =
||      1     2
||      3     4
>> x(:,1)
|| ans =
||      1
||      3
>> x(:,2)
|| ans =
||      2
||      4
>> x(1,:)
|| ans =
||      1     2
>> x(2,:)
|| ans =
||      3     4
```

You may do numeric operations on matrices. Adding and subtracting matrices adds and subtracts each corresponding element of a matrix. If you wish to do the same for multiplication, division, and exponentiation, you must use the "dot operators." The dot operators are `.*`, `./`, and `.^`. These to element by element operations on the matrices. Of course, to do any of these operations, you need to have matrices of equal dimensions. Here is an example of each operator:

```
>> a = [1,1;1,1];
```

```
>> b = [1,2;3,4];
>> a+b
|| ans =
||     2     3
||     4     5
>> a-b
|| ans =
||     0    -1
||    -2    -3
>> a.*b
|| ans =
||     1     2
||     3     4
>> a./b
|| ans =
||    1.0000    0.5000
||    0.3333    0.2500
```

You can also generate monotonically increasing matrices very easily using the syntax `matrix=startnumber:stepsize:stopnumber`, like so:

```
>> t=0:1:5
t =
    0     1     2     3     4     5
```

Finally, you may also concatenate matrices. This is a very straightforward process; you simply add the matrices that you want to concatenate together as elements of the bigger, final matrix. Here is an example:

```
>> a = [1;2;3];
>> b = [4;5;6];
>> c = [a,b]
|| c =
||     1     4
||     2     5
||     3     6
>> d = [a;b]
|| d =
||     1
||     2
||     3
||     4
||     5
||     6
```

You may also do everything in one step, like so:

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

That's all you need to know about matrices at the moment; onto M-Files.

Section 2.4: M-Files

M-Files are used for scripting in MATLAB. Now that you have been introduced to the command window, over half the work of understanding the scripting interface in MATLAB is finished. Basically, an M-File is simply a text file that has all of the commands that you wish to execute, separated by either a return/line-feed or by semi-colons. The percent sign (%) designates the area beginning at the percent sign to the end of a line as a comment.

Here is an example of an M-File:

Example 2.4.1

```
%BasicMFile

%This M-File takes a number that you specify, num1, and passes it as an
%argument to the peaks function, and then does a surface plot of the result

num1 = input('Please input desired matrix dimensions: ');

surf(peaks(num1))
```

This example would work exactly the same if you typed all of the non-comment commands into the command window. `surf` is the surface plot function that is built into MATLAB, and `peaks` is a sample three dimensional function that plugs nicely into `surf`. We will have a more detailed discussion of the `surf` function in the next section.

Right now, I want to talk about the `input` function. The `input` function displays a message, that you specify as an argument for the function, in the command window, and waits for the user to input data. When the user inputs data and hits return, the `input` function returns the value that the user put in. In the M-File above, we assigned the user's input to a variable called `num1`, which we then passed to the `peaks` function. The input function simply provides a way to make your M-Files more dynamic.

You can use the input function to provide the familiar "Press any key to continue" functionality in a program also. Simply make the message whatever you like, and ignore the output.

Another useful thing to do in an M-File is to make a for loop. While you can do these in the command window as well, they are much more useful in the context of a script. In MATLAB, the syntax for a for loop is very simple:

```
for iteration_variable=startindex:stepsize:endindex
    <do some stuff>
end
```

Here is a simple example that counts from 1 to 10 for you:

Example 2.4.2

```
%Ex-2.4.2

for i=1:1:10
    i
end
```

This example simply prints the iteration variable once every iteration.

While loops are very similar to for loops as far as syntax goes. The syntax for a while loop is as follows:

```
while <condition>
     <insert commands here>
end
```

Here is an example:

Example 2.4.3

```
%Ex_2_4_3

i = 1;
while i <= 10
    i
    i=i+1;
end
```

This example does the same thing that Example 2.4.2 did, only it does it with a while loop.

Finally, you may loop through different elements of matrices by doing the following:

Example 2.4.4

```
%Ex_2_4_4

a = input('Input any n x n matrix: ');

SIZE = size(a);

for i = 1:SIZE(1)
    a(i,:)
end
```

This example iterates through each row, displaying each one. The `size` function accepts a matrix as an argument, and it returns the size of that matrix as a 1 x n matrix, where n is the number of dimensions of the argument. You can use the `size` function to dynamically loop through matrices of different sizes.

Section 2.5: Visualizing Data

MATLAB offers a great many ways to go about visualizing data. You can do all kinds of basic 2d plots, as well many 3d plots. We will mostly be focusing on 2d Cartesian plots, 3d surface plots, 3d contour plots, and 2d contour plots. Here is a quick reference for the main visualization functions present in MATLAB.

2d Cartesian: `plot`

2d contour: `contour`

3d surface: `surf`

3d contour: `contour3`

Each of these functions have many levels of functionality; we will only be covering the basics. If you want to understand the more fancy things that they can do, check out the MATLAB help file entries for these functions for more detail.

First, we will start with the simplest one of all of these: the `plot` function. The plot function, in its most basic form, simply accepts an x array and y array of the same size and pairs them. If you want to plot a function--say, $f(x) = x^2$, you would need to generate a matrix that has all the x values in it, and then proceed to use the `.^` operator on that matrix to generate the y values. Here is an example that does just that:

Example 2.5.1

```
%Ex_2_5_1

x = -4:.1:4;
y = x.^2;

plot(x,y)
```

If you run this M-File, you will see a nice graph of $f(x) = x^2$ pop up. You can do this with any sort of function you wish. If you want to change the way that the line looks from the default of a solid blue line to, say, a series of red circles with red lines connecting them, you would do the following:

Example 2.5.2

```
%Ex_2_5_2

x = -4:.1:4;
y = x.^2;

plot(x,y,'r-o')
```

This is just one of the many formatting choices that you have; I just wanted to let you know that you have the option of changing how it looks. There are many, many more options. Just look in the documentation for more information on how to go about formatting your graph as you would like it. Contact me if you can't find the information, and I'll point you in the right direction.

That's about it for the `plot` function; onto the 3-dimensional plots. We'll start with the contour plot, since that is the simplest. Each of the dimensional plots require you to pass them two square matrices that represent the x and y values. The functions then match up the x and y values element, and couple those to a third z matrix. If you are plotting a function, the easiest way to go about doing this is to use the `meshgrid` function. This function takes two separate 1 x n matrices (n must be the same for both matrices), replicates the first across n columns, the second across n rows, and outputs both resultant n x n matrices. It's much easier to show you than explain it, so here you go:

Example 2.5.3

```
%Ex_2_5_3

x = [1,2,3,4];
y = [5,6,7,8];

[X, Y] = meshgrid(x,y)
```

The output of this M-File looks like this:

```
X =
     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
Y =
     5     5     5     5
     6     6     6     6
     7     7     7     7
     8     8     8     8
```

As you can see, this allows you to get every possible combination of x and y, which is your goal. These new matrices may then be acted upon to calculate a z matrix that you can plot with a 3d plotting function.

It's not generally useful to plot functions by manually creating matrices that include the points that you wish to plot. Thus, we will generally be using the syntax introduced in Section 2.3 that generates monotonically increasing matrices to plot 3d functions. Here is an example that plots $f(x,y) = x^2 + y^2$:

Example 2.5.4

```
%Ex_2_5_4

x = -4:.1:4;
y = -4:.1:4;

[X, Y] = meshgrid(x,y)

Z = X.^2 + Y.^2;

contour(X,Y,Z)
```

You'll see a nice contour plot pop up in the figure window; that's all there is to it.

You do the same exact thing with all of the other 3-dimensional plotting functions. Everything is the same, as that last example, except you change `contour` to whatever 3-dimensional graphing function that you would like. Like the `plot` function, there are many other ways to format the graph by passing arguments to the function. If you want to know how, either refer to the documentation or contact me.

There are many other visualization functions available to you through MATLAB, including histograms, vector fields (which we will be dealing with in the experiment itself), 3d vector fields, wire frame, 3d discrete points, etc. We will not cover these here, since they are not applicable to the lab that we are about to get to; refer to the documentation if you wish to learn about them.

Section 2.6: Importing Data

Importing data into MATLAB is very simple, so this will be a short section. There are two main ways to do it; you can use the import dialog, or you can use the `dlmread` function. Both require you to have your data in some standard ASCII spreadsheet format, such as tab separated or comma separated (as we discussed a little bit in the LabVIEW section).

The import dialog may be called from the Workspace window. It is a button at the top that looks like a folder with an arrow pointing out of it. If you click this, you will be presented with a wizard-type dialog that walks you through all the necessary steps of importing data.

The import dialog is useful if you just wish to do your work using the command window, but if you wish to script whatever you are doing, it is not so useful. The best thing to use in scripting situations is the `dlmread` function. This function accepts a file path as its first argument and the delimiter as the second argument. It then outputs a single matrix that represents whatever data it read. Make sure that whatever you're reading doesn't have anything but numbers in it; if it runs into something else, it will give an error.

To test the `dlmread` function, open up notepad (either find it buried wherever in the start menu, or go to Start > Run and type "notepad"), and create some sort of tab-separated data. Tab-separated data simply means that you designate each new column by a tab and each new row by a return. Alternatively, if you don't want to create it directly, you could open up Excel, type some data in, and tell it to save as tab-separated values.

Once you've created your data, do the following in the command window:

```
>> data = dlmread('C:\Your\file\path\yourfile.txt', '\t')
|| <MATLAB outputs whatever the contents of your file was>
```

It's that simple. You can script it also, of course (which is generally more useful). The only other important thing to remember is that you must find some way to get your data into an ASCII standard file (text file in other words); MATLAB doesn't (without an add-on anyways) support proprietary formats such as XLS. You can get it to work with Excel if you really want to, but you have to know what you are doing, and it is generally not worth the effort. If you really want to know how, search for "Excel" in the documentation.

Section 2.7: Closing Comments

As you have hopefully seen from what we have done so far, MATLAB is a very useful tool for data analysis. Once again, I have only scratched the surface of its ability; to really understand it, you need to just play around with it. Hopefully you are now familiar enough with it to understand the documentation, which is really the most important thing for our purposes, as far as the lab is concerned.

I didn't bother putting these examples up on the web site, since they were all so simple; however, if you would like them to be up, I do have them, and it wouldn't take very long to put them up. You will have considerably more MATLAB content, that is a little more complicated, available to you for use in the actual lab exercise that begins on the next page.

Feel free to contact me with any questions or comments that you may have; I will be more than happy to assist you any way that I can.

Lab

Section 3.1: Introduction

This lab's goal is to show you how to get computers to do the majority of your work for you. It will be setup a little different than normal, in that you are given free reign to do whatever you would like to do to reach your goal. What goal? Good question.

Goal: Use conducting paper, a SummaSketchIII, a power supply, a computer with LabVIEW and MATLAB, a NI USB-6008 DAQ, and physics to describe the electric field and potential present on the conducting when the power supply is hooked up to the two electrodes on the conducting paper.

The physics behind this experiment is purposefully very simple; the focus is on making your own experiment to show the results that you already know the answer (in great detail) to. In the following pages, I will be presenting you with information on the various aspects of the equipment you are using, as well as things in LabVIEW and MATLAB that I think will come in particularly handy. It is then up to you to put those things together to come up with something useful.

Because this is so open ended, and hence easy to get lost in details, I will be happy to provide any sort of help that I can. In fact, I will try to be in class fairly often, and if you ever request me to be there sometime during class, I will do my best to be there.

Section 3.2: Equipment

There are two key components to this experiment: the SummaSketchIII and the NI USB-6008. I will begin by explaining what the SummaSketchIII does, and how we will be using it.

The SummaSketchIII is a digitizer pad that has a modified stylus that allows it to measure voltages with its metal tip. We will be using the pad to allow us to simultaneously measure coordinates and couple those coordinates to a voltage. To do this, we will be placing the conducting paper on top of the SummaSketchIII and taking our measurements that way.

To use the SummaSketchIII, you simply touch the stylus to where you want to measure, and press down on the pen itself or on the little blue button on the top of it. This causes the SummaSketchIII to report data. This data is a short stream of bytes that is relatively complicated to explain. Luckily, you won't have to bother with that aspect of things; I have already written a driver that will handle it. All you have to do is drop my little SubVI on the block diagram, and voila--out pops coordinates whenever you need them. It is then your job to synchronize these coordinates with the voltage that is read though the DAQ.

To use the driver that I wrote, refer to the web site. There, you can find the documentation as well as the SubVI itself. It is under the "Lab Resources" area of the "Introduction to Modern Data Acquisition Lab" page.

The data acquisition device that we will be using is the NI USB-6008. It is basically a fancy, computer-controlled volt-meter. Thankfully, as we saw in the LabVIEW section on data acquisition, NI makes it really easy to control their instruments through LabVIEW using the DAQ-Assistant SubVI (see Section 1.8).

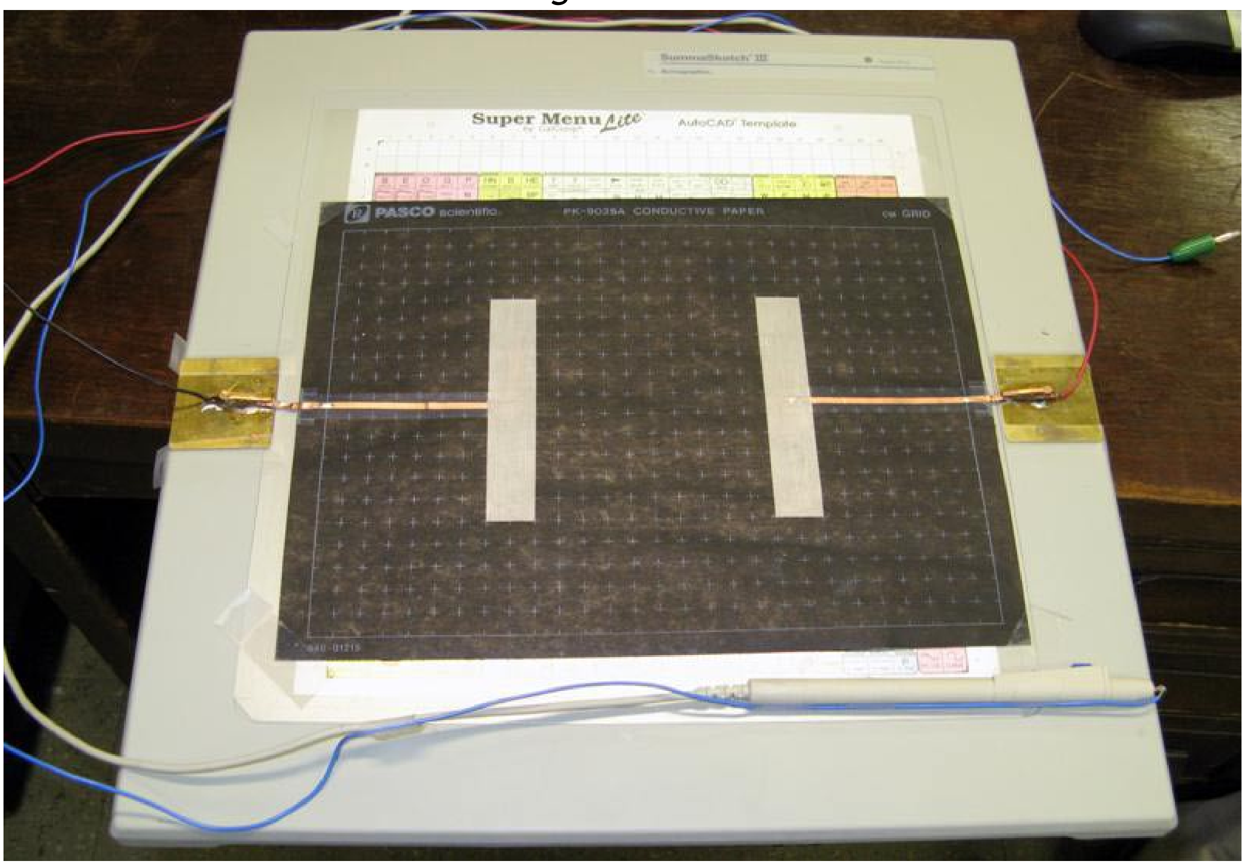
The USB-6008 will be used to measure the voltage that is present at the tip of the SummaSketchIII stylus. This will allows us, through LabVIEW, to take the different coordinates associated with physical locations on the conducting paper and assign a measured voltage to that position. With that data, you can proceed to do lots of interesting things.

Section 3.3: Goals

You already know the overall goal: to measure the voltage and electric field at multiple points on a piece of conducting paper. This goal may be further broken in to multiple “sub-goals”; I will proceed to do that now.

First off, you must plug in everything properly. Here is a picture of what the SummaSketchIII board should look like with the conducting paper, alligator clips, etc. hooked up.

Figure 3.3.1



Basically, all you must do is affix the conducting paper and alligator clips to the SummaS-ketchIII. You then connect the alligator clips to the power supply (set it to somewhere between +5V to +10V), and if you desire, patch the negative terminal to the ground terminal of the power supply. Since we will be using differential measurements with the DAQ, the ground patch shouldn’t really matter, but you may do it if you would like.

Finally, you will want to hook the lead that comes out of the stylus to the positive terminal of the USB-6008, and the negative terminal of the USB-6008 should be patched to the negative termi-nal of the power supply (or ground if you patched the ground to the negative terminal). To make things simple, I would suggest using channel ai0 of the DAQ, but you may use whatever you wish to use.

Now you will want to test your setup. There are several ways you can test it, but the simplest would probably be to turn the power supply on, navigate to MAX, run the test panels for the DAQ, and touch a few places on the conducting paper to get logical results. You may also want to directly start my "SummaSketchIII Get Coordinates" VI and see if the computer is getting coordinates properly. Do whatever testing you think is proper.

After all the equipment is properly setup, its time to begin constructing the program. Here is a breakdown of the core steps that you must take to take the measurement that we are looking for:

- Create a monitoring loop that waits until there are coordinates to be read from the SummaSketchIII (using the "Anything to Read?" output of my driver)
- Make this monitoring loop terminate when it finds that there are coordinates to read
- When the monitoring loop terminates, immediately read a voltage from the DAQ
- Find some way to store the results while you're taking the data (either pop it into an array or append it to a file)
- Put all of the above in a loop so that you can take multiple points
- Export the data to a file for storage
- Possibly use a MATLAB node to feed the data directly into MATLAB (that's up to you to decide)
- Use the data that you take to find out the electric field using whatever tools you wish (MATLAB is suggested)
- Do whatever you wish as far as displaying the results (plots, tables, histograms, whatever)

That's it! Now, in the next section, I will discuss a specifics about LabVIEW and MATLAB that may be useful to you.

Section 3.4: MATLAB and LabVIEW Tips

First, I will begin by describing how to go about doing one of the integral parts of designing this experiment: reading the coordinates. I will begin by describing a little bit about how the tablet works.

There are two main modes for the SummaSketchIII (from now on, I will refer to it as the SSIII): Stream Mode and Point Mode. Stream mode constantly reports data whenever the stylus is close enough to the pad to produce any data. It reports data at 30 reports per second by default, but you can change that behavior fairly easily. My driver will handle this mode, but it's not very useful given what we wish to use the SSIII for; we need point mode for our application. Point mode only takes data once for every push of the button or depression of the stylus. You can set the mode to point mode simply by running the VI entitled "SSIII - Set Point Mode." See the documentation on how to use that VI (it's very simple).

After you set the point mode, you may wish to configure the resolution of the pad. Here is a list of the possible resolutions:

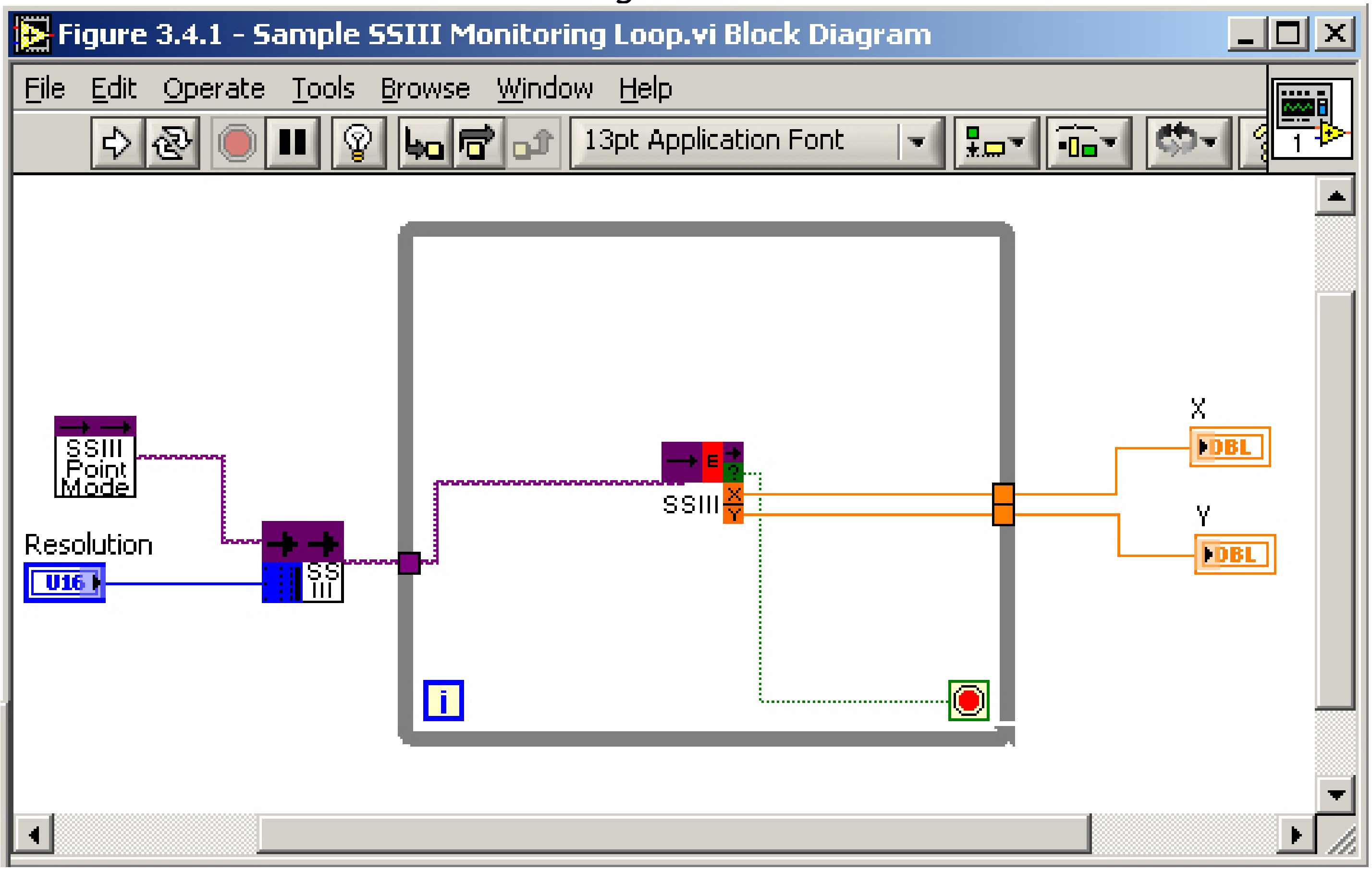
- 1 lpi
- 2 lpi
- 4 lpi
- 100 lpi
- 200 lpi
- 10 lpmm
- 400 lpi
- 500 lpi
- 20 lpmm
- 1000 lpi
- 40 lpmm

The resolution doesn't matter too much as far as accuracy is concerned; however, it is essential to know what the numbers that are being reported as coordinates mean. You must scale the dimensions according to what resolution that you select. This is essential if you wish to report the field in standard units of any kind.

After you have set the tablet to point mode and set the resolution, you are ready to take coordinate data. You use my VI called "SSIII - Get Coordinates" for that task. Once again, read the documentation to see how to go about using that VI.

Here is a way that you might go about taking data with the SSIII.

Figure 3.4.1



This VI begins by setting the SSIII to point mode. It then sets the resolution to whatever resolution is specified on the front panel. Finally, it enters a loop that terminates when the SSIII Driver reports that there are coordinates to read, and it outputs those coordinates when the loop terminates.

It could be made better by making use of the error reporting feature of the driver. For example, if an error is reported, you could make it so that the loop doesn't terminate and the serial buffer gets cleared. This lets you start over with a clean slate. Also, you could add a stop button, or any other set of features that you desire; hopefully you get the picture.

Now, instead of giving a detailed tutorial on how to do the contour plot with the vector plot superimposed upon it, I will present it in generalized steps and leave the particulars up to you and the MATLAB help function.

First, I will present you with the problem. As you saw in section 2.5, you have to have a grid of X and Y coordinates and you must have a Z value paired with every possible combination of them. In other words, if you have X = [1 2 3 4] and Y = [5 6 7 8], you need a z data point for (1,5), (1,6), (1,7), (1, 8), (2,5), (2, 6)... etc. But what do you do if you only have discrete points? Lets say you take an X-Y coordinate pair off of the SSIII, and then you assign a voltage to that coordinate. When you do that multiple times, you'll get lots of points that are not arranged in a grid pattern. For example, you might have one X-Y coordinate that is (3,6) and then another that is (5,19). In order to have it in grid form for plotting in MATLAB, you would need to have the points (3,19) and (5,6) as well. But you won't have those points. So our task now is to find a fancy way to plot it anyways.

Here are the steps:

- Take your data in the form of 3 vectors, X (x-coordinate), Y, (y-coordinate) and V (voltage)

- Use MATLAB's `linspace` function to generate a monotonically increasing set X Y data that starts at the minimum of your data's X and Y coordinates (use `min` and `max` to find the max and min of the coordinate data)
- Plug this new data into an interpolating function called `griddata` that will generate V values for the intermediate values; this will result in you having interpolated voltages associated with the linspace coordinates
- `meshgrid` the results
- Use the `gradient` function to take the gradient of the Voltage to get the electric field
- Evaluate the gradient at the `linspace` generated points
- Use `contour` to plot the original data
- Turn `hold` on so that you can put two plots on top of one another
- Use `quiver` to generate the vector (quiver) plot
- You should then have a contour plot that has the electric field vectors on it
- Turn `hold` off
- Do a `new figure` so you can pop up a new window
- Pass `surf` the `meshgrid`'ed `linspace` coordinates and the `griddata` voltages associated with them
- Voila, you're done

If you come up with any other big questions, feel free to ask me, and I will probably add them to this section.

Section 3.5: Closing Comments

Now it looks like you're on your own! Feel free to be creative with anything that you can think of to do with this. The things I've mentioned here are just the things that I thought of that you might want to do with something like this; there are many other possibilities. In fact, this was born out of a desire to rework the undergraduate lab on this subject, so if you can come up with any good, new ideas, by all means, go for it. A version of your idea may find its way into the undergraduate curriculum.

If you wish to ask me questions, I'll do my best to be in the lab on most days that you are. If not, I would be happy to arrange a time to meet with you outside of class; just let me know when would be a good time.