LabVIEW for Control Analysis and Design

Marian Chaffe and Matanya Horowitz
Based on the course taught by Todd Murphey
0.1 Introduction

LabVIEW is a graphical programming language in principle capable of the same utility that programming in C or C++ can provide. Some capabilities of C++ are more difficult to obtain, but for the purposes of control systems—the focus of this short tutorial—LabVIEW is an exceptionally convenient programming language. This is for two primary reasons:

1. LabVIEW enables programming that mirrors that graphical analysis tools (such as block diagrams) that we use to analyze control systems;

2. LabVIEW seamlessly (well, at least ideally seamlessly) incorporates “hardware-in-the-loop” needs into code.

We will see both of these become evident over the course of this tutorial.

0.1.1 Background and Purpose

This tutorial is being written as an accompaniment to the control laboratory course ECEN 4638 at the University of Colorado, based on student comments from previous laboratories. Hence, this tutorial is not intended to be exhaustive—exhaustive tutorials have not been very beneficial to students because they tend to be overwhelming. Instead, this tutorial aims to very specifically introduce students to LabVIEW in such a way that they can use it in specific application to control systems and so that they may become comfortable both with LabVIEW in particular and graphical programming in general, both of which are becoming standards in industry. Moreover, this type of exposure is intended to encourage more student investigation (both in LabVIEW, controls, and life in general) rather than explicitly laying out the solutions to all anticipated problems.

0.1.2 Philosophical relationship between LabVIEW and other programming languages.

LabVIEW is to C (C++, Fortran, etc) as C is to assembly. More to come here.

Data Flow, blocks that represent data manipulation, graphical, similar to block diagrams, lines represent data types,

Lastly, it is worth mentioning that LabVIEW is compiled, not interpreted. Among other things, this often means that when you are using new functionality, it will require some time for it to compile all the code, particularly the code that references hardware on your machine. Be patient!
0.1.3 Organization
This tutorial will be organized according to the needs of an introductory controls course. The goal is not to systematically introduce students to all of LabVIEW’s capabilities. It is the author’s view that this is simply an untenable goal leading to certain failure and confusion on the part of the students. Instead, we will focus on using simple examples, as they become useful, to illustrate new tools being used within LabVIEW. As the tutorial proceeds, it is assumed that less and less explicit instruction will be required, so some hints may simply lead the reader to look at a particular tool on his own and merely indicate that it is indeed a useful tool.

0.2 Basics of Graphical Programming
Representing complex systems as block diagrams is already a common use of graphical representations. Textbooks use this regularly in control systems design, and the idea is largely to replace a mathematical representation of the entire system with a series of subcomponents connected together by various types of interconnections. This is the fundamental concept behind graphical programming as well, making it modular in design and flexible in most applications. Although there are certain types of computations that we will encounter where graphical programming is not a good choice, it will be largely to our advantage to program in this way.

0.3 Graphical Programming as Data Flow
Consider the following typical type of simple programming problem. Given a temperature in Fahrenheit $F^\circ$, we want to compute the Celsius degrees $C^\circ$. We want to use the formula

$$F = (212 - 32)/100C + 32$$

and

$$C = 100/(212 - 32)F - 32.$$

as expressions. In standard text languages, this might look something like:
float celsius(float fahrenheit);

main()
{
    float DegreeF=76;
    float DegreeC=celsius(DegreeF);
    printf("The Temperature is \%f F and \%f C\n", DegreeF, DegreeC);
}

float celsius(float fahrenheit)
{
    float celsius = (5.0/9.0)*(fahrenheit-32);
    return celsius;
}

Figure 1: A LabVIEW-related graphic

The blocks that will be required for this circuit are in the Programming // Numeric Palette and should be placed within the Block Diagram. To place the user interface components, you may place a numeric constant, right click and change it to either Indicator or Control. You may also place a numeric indicator or control in the front panel. Build this circuit and test it.

0.4 Simulation in LabVIEW

The “simulation loop” will be one of the fundamental tools used in both simulating systems and running hardware-in-the-loop experiments. It can be obtained by opening up the Simulation (or possibly the Control Design & Simulation) palette in the back panel (shown
in Fig. 2, clicking on the upper left “simulation loop” icon, and dragging it into the back panel. It may then be stretched to any desired size. The result will look something like the loop seen in Fig. 3. All the things that need to execute at every time step in a simulation or hardware experiment should be placed within the simulation loop. Everything else can be kept outside of it.

The simulation loop determines the

1. integration algorithm used (e.g., Euler (fixed time step), Runge-Kutta, and variable time step algorithms);
2. Determines the step size both for fixed time step integration algorithms and for running hardware experiments;

3. length of time for the simulation;

4. timing parameters (involving the clock, an external clock, how fast to access said clock, etcetera).

All these options can be configured by right-clicking on the simulation loop and selecting Configure Simulation Parameters. The dialog box is shown in Fig. 4.

![Figure 4: A LabVIEW Simulation Loop](image)

The palettes contain the other operations you may wish to use. These include:

1. Generating a signal. For instance, one can create a step-input from the Signal Generation palette. After placing the step-function in the Simulation Loop, double click on it. This will bring up the configuration window. Within the configuration window is a list of parameters. Clicking on any of the parameters will open “Parameter Information,” which allows you to modify the selected parameter (i.e. final value of step-input = 10).

2. Mathematical operations, such as summation and multiplying a signal by a gain, can be found in the Signal Arithmetic palette. Each of these can be individually configured—for instance, clicking on the summation block in the panel will allow one to configure it for “negative” feedback.
3. Linear effects, such as time delay are found in the Linear Systems palette. (Note that within LabVIEW, a time delay is known as “transport delay.”)

4. Nonlinear effects such as saturation are found in the Nonlinear Systems palette.

5. Graphics: Plots can be generated using Graph Utilities. In general, you will want to use a sim-time wave form, but you may wish to use an XY graph if you want trace functionality.

Figure 5: A simple feedback controller

0.4.1 Creating Sub-Systems in your Simulation

You will find that your code gets quite complicated as you get more functionality. If you want to replace part of your code with a subsystem block, just select the parts you want in the subsystem and go to Edit and select Create Simulation Subsystem. This will create a block that has the same inputs and outputs and the region you selected. You can then view the contents of the block by right clicking on it and selecting Open Subsystem.

0.5 MathScript—a command line utility

MathScript has much of the functionality of other command line scripting languages such as Matlab and MatrixX. Standard commands useful to control design include:

1. \texttt{tf} creates a transfer function;

2. \texttt{step} plots the step response of a transfer function or state space system;

3. \texttt{bode} plots the bode plot of a transfer function;

4. \texttt{rlocus} plots the root locus of a transfer function;
5. *nyquist* plots the nyquist plot of a transfer function.

Example code that might be illuminating includes:

Here are some different ways of doing Example 2.1 in the FPE textbook. You can get a step response entirely numerically, like the book does:

```matlab
num=1/1000;
den=[1 50/1000];
sys=tf(num*500,den);
step(sys)
```

You can use variables:

```matlab
m=1000;
b=50;
num=1/m;
den=[1 b/m];
sys=tf(num*500,den)
```

You can define the transfer function as a fraction (i.e., the way you would write it down). This is ultimately going to be preferrable.

```matlab
s=tf('s');
g=(500/m)/(s+b/m)
step(g)
```

Lastly, you can incorporate “.m” file code into your LabVIEW VI by using the MathScript block from the Structures palette. An example of this is given in Fig. 6. In order to do this, one must create the MathScript block, write the desired code, create an output of the block (by right clicking on the edge of the block), and finally define that output to be of the appropriate data type (in the case of Fig. 6 a “TF Object”, and in the case of Figs. 7, 8 and 9 a “SS Object”).

### 0.6 Graphs in LabVIEW Simulation Module

Graphs are typically created using a sim-time waveform (in the simulation module), as seen in Fig. 10. However, if you want trace functionality, the you should use the XY Graph (also in the simulation module). The back panel for this is seen in Fig. 11. Note that the XY Graph requires both time (which we generate in this case using a ramp function) and the actual signal, which are then plotted against each other. In order to plot them both, one must “bundle” the two signals together using a bundle block (located in the “Cluster and Variant” palette).
0.6. GRAPHS IN LABVIEW SIMULATION MODULE

Figure 6: MathScript defining a transfer function in the simulation module

```
a=1;
b=2;
s=tf('s');
g=(a)/(s+b);
```

Figure 7: MathScript defining a state space system in the simulation module

```
a=[0 1; -1 -1];
b=[0; 1];
c=[1 0];
d=[0];
sys=ss(a,b,c,d);
```

Figure 8: MathScript defining a state space system, converting it to a transfer function, and using it in the simulation module

```
m=1;
b=1;
k=0;
a=[k; 1; -m -b];
b=[0; 1];
c=[1 0];
d=[0];
[e, f] = ss2tf(a, b, c, d);
sys=tf(e, f);
```
Figure 9: MathScript defining a degenerate state space system, converting it to a transfer function, and using it in the simulation module

m=0;
b=0;
k=-1;
a = [k 1; -m -b];
b = [0; 1];
c = [1 0];
d = [0];
[e, f] = ss2tf(a, b, c, d);
sys=tf(e,f);

Figure 10: A Sim-Time Waveform

The cursor is created in the front panel (see Fig. 12) by right clicking on the graph and selecting *Properties*. Then select cursor, choose to add a cursor, and a cursor will show up on the graph. Note that the cursor can only select actual data points—it will not interpolate.
Therefore, you may wish to select a maximum step size in your integration algorithm that makes it move smoothly from point to point. (E.g., reduce the maximum step size.)

![An XY Graph Back Panel](image)

**Figure 11: An XY Graph Back Panel**

### 0.6.1 Some Notes on Graphing in LabVIEW

1. There are two distinct types of XY Graphs. One is the Simulation Module “Buffer XY graph” (found in the Simulation Module Palette under “Graph Utilities”) that can only be used in the simulation module. The other is the standard XY graph that can be found under the “Graph” pallette of the front panel. (I.e., you find it by right-clicking on the front panel, select the “Graph” palette, and then click on “XY Graph.”)

2. The default amount of data a Waveform Chart in the Simulation Module displays is 1024. If your time steps are too small in a simulation or experiment, you may not get all your data. Hence, if this happens you should change the “Chart History Length” to a larger number (by right clicking on the graph).

3. Lastly, the time axis sometimes gets set to Day/Year format. If this happens, go into “Properties” and under “Format and Precision” set Type to SI notation.

### 0.7 LabVIEW Saving and Reading

This tutorial is to give you a sense of how to save and read data within LabVIEW. Saving data is reasonably straight forward. You create a VI, and use a “Collector” to collect all the data during the simulation/experiment. Then you must use “Unbundle By Name” and put all the data you want to save into the same array by using “Build Array.” Lastly, the “Write File To Spreadsheet” utility will allow you to write the file in a convenient format. (All this is shown in Fig[13]) Note that you must allow the simulation to finish or data will...
not be written to the file. Also note that LabVIEW allows you to plot multiple plots on the same graph, as shown.

Now, reading data is similarly straightforward. Assuming that you have saved the data as above (in a spreadsheet format), you can use the “Read From Spreadsheet” utility. The data will be in an array format already, but you must use the “Index Array” to select the data. Figure 14 shows the use of the “Index Array” block to select the first ("0"th) and second ("1"st) signals from the data. These can then either be used or bundled together and plotted.
Figure 13: (top) A VI that saves data from an FPGA, (bottom) an enlargement of the section of code that saves data.

The collector can be found under Simulation // Utilities, while the Write To Spreadsheet VI can be found under Programming // File I/O. Modify your code from Figure 8 to save the waveform. Then import this waveform using the code from Figure 14.
Figure 14: A VI that reads data