MathScript 3

ELECTRICAL ENGINEERING 20N
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

SIMON HONG, HSIN-I LIU, JONATHAN KOTKER, HOWARD LEI, AND BABAK AYAZIFAR

1 Introduction

In this lab, we will explore `MathScript`, the text-based scripting language of LabVIEW. `MathScript` is a very powerful tool, and is useful in dealing with arrays and signals, especially those not grounded in the time domain. `MathScript` involves text-based programming, which is in contrast to the graphical dataflow programming presented in the previous two lab sessions. As we work through this lab, we will find various advantages and disadvantages to both methods, but the same functionalities can usually be achieved in either method. Eventually, however, complex VIs involve a successful mix of both methods, and fortunately, `MathScript` code can be directly integrated into LabVIEW block diagrams through `MathScript` nodes.

1.1 Configuring your Local Machine

Please make sure that you have installed `MathScript` on your local machine (*not* on the instructional machines). In order to install `MathScript`, insert your LabVIEW DVD and run the LabVIEW setup once again. In the `Product List` screen, select `LabVIEW 2009 MathScriptRT Module` for installation, and rerun the installation.

1.2 Lab Goals

- Explore the `MathScript` environment.
- Integrate `MathScript` code into block diagrams using `MathScript` nodes.
- Implement LabVIEW constructs in complex VIs.

1.3 Checkoff Points

2. `MathScript Window` Environment . . . . (30 minutes)
   1. Example VI . . . .
   2. Self-Exercises . . . . (10%)
3. Creating a `MathScript` . . . . (30 minutes)

1. Example VI . . . .
2. Self-Exercises . . . . (10%)

4. MathScript Nodes . . . . (45 minutes)

1. Example VI . . . .
2. Data Types of Output Terminals . . . .
3. Self-Exercise . . . . (10%)

5. Miscellaneous Exercises . . . . (45 minutes, 20%)

6. Acknowledgments . . . .

2 MathScript Window Environment

2.1 Example VI

The following exercise will introduce you to the MathScript Window and its different components.

1. Open the LabVIEW MathScript Window by going into Tools → MathScript Window from the LabVIEW environment.

2. Initialize an array from 1 to 10.

   (a) The MathScript Window includes a Command Window in the lower-left corner. Enter the following command in the Command Window and press the Enter key:

   ```
   t = 1:10
   ```

   (b) Notice that the result of the command you entered appears in the Output Window, located just above the Command Window. The result of this command is a vector of ten elements, where the elements start at 1 and end at 10, with a step size of 1.

   > For the arrays in MathScript, array positions begin at 1, whereas for the arrays created in LabVIEW, array positions begin at 0. *This is a common cause of error*. Also, notice the general structure of commands that declare arrays in LabVIEW: start_value:step_size:end_value. If the third field is not specified, the step_size is assumed to be 1 by default, and the second field is assumed to be the end_value.

3. Examine the Variables tab and the Preview pane.

   (a) Locate the tabs labeled Variables, Script, and History at the top right of the LabVIEW MathScript Window.

   (b) Click the Variables tab to display the Variables page. This page displays a list of all variables you define and previews variables that you select.

   (c) Notice that the variable list contains an entry for the t vector you defined in step 2a. Click the t in the variable list to display the contents of t in the Preview Pane located in the lower right corner of the LabVIEW MathScript Window.

   > You can also get a quick summary of all variables available, as well as their contents, by entering the whos command in the Command Window.

4. Generate one period of a sine wave.

   (a) Enter the following commands in the Command Window. Press the Enter key at the end of each line.

```
t = 0:0.1:2*pi;
y = sin(t);
```

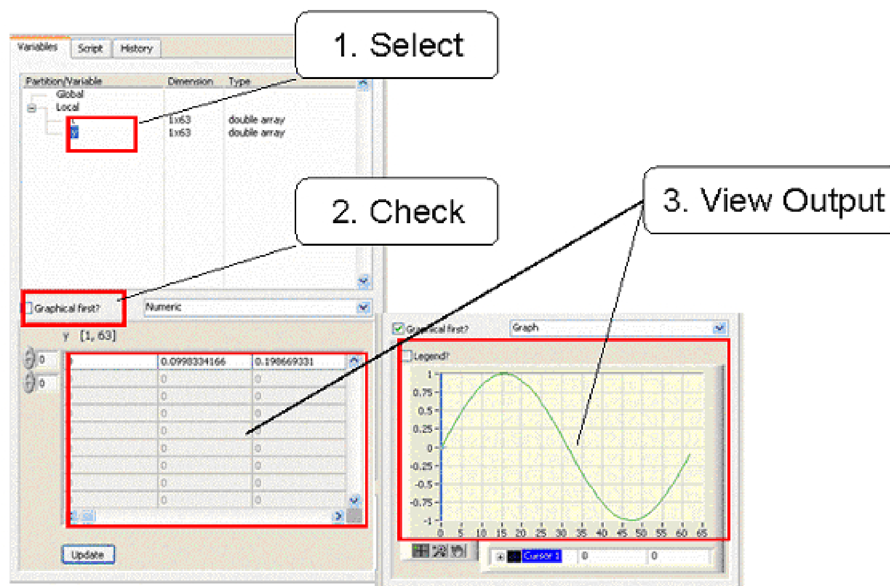
(b) After you press the `Enter` key, LabVIEW displays each command in the `Output Window`. LabVIEW does not display the output for each command, because the semicolon at the end of each line directs LabVIEW to suppress the output.

> Not only does the semicolon suppress the output, it also speeds up processing, because output no longer has to be echoed to the screen. If you do need to show some output, be sure not to put a semicolon at the end of the line that produces the output.

5. Graph the variable `y`.

   (a) On the `Variables` page, click the local variable `y`. Then modify the view options on the `Variables` page to view the contents of the `y` variable as a graph or as numerical elements, as shown in Figure 1. You can also use these different views to modify the contents of the variable.

Figure 1 Examining Variables on a Graph.



2.2 Self-Exercises

1. `MathScript` can handle element-wise computation of two vectors/arrays. The `.` (dot) operator, followed by a scalar operation of interest (such as `*`, `/`, and `^`), will apply the operation *pointwise*. For instance, enter the following command into the `Command Window` and observe the result:

```
array1 = [1 2 3 4].*[1 2 3 4]
```

We can also multiply a scalar with a vector using the pointwise operator `.*`, in order to multiply the scalar with each element in the vector:

```
array2 = 2.*[1 2 3 4]
```

Try initializing `array1` with the same arrays that you initialized it with earlier, except using the `.^` operator instead of the `.*` operator. What do you observe?

2. `MathScript` also has help documentation written up for most of its built-in commands. These can be accessed through the `help` command. For instance, determine the purpose of the `length` command by typing `help length` in the `Command Window`, and use it to find the length of `array2` defined in step 1.

3. Using the `MathScript Window`, identify the dimensions of the result generated by each of the following commands using the `size` command, and determine if the result is a scalar, a row vector, a column vector, or a matrix.

   (a) `array1 = 1 : 5`

   (b) `array2 = 1 : 1 : 5`

   (c) `array3 = 1 : -1 : -5`

   (d) `array4 = 1 : 1 : -5`

   (e) `array5 = [1 2 3 4 5]`

   (f) `array6 = [1 2; 3 4]`

   (g) `array7 = [1; 2; 3; 4]`

   Should you want to completely eliminate a variable and start afresh, you can use the `clear` command. The `clear` command, followed by the name of the variable, will remove that variable from memory and further access. **If the `clear` command is not provided with any arguments, it removes *all* variables, so be careful!**

4. How is the `size` command different from the `length` command?

5. From step 1, we know that we can multiply two vectors pointwise, or a scalar with a vector, using the `.*` operator. In this exercise, we will examine what happens when the `*` operator is used instead. Enter the following commands and determine whether or not an error arises. If an error does not arise, explain the output.

   (a) `a = 2 * 4`

   (b) `b = 2 * [1 2 3 4]`

   (c) `c = [1 2 3 4] * [1 2 3 4]`

   (d) `d = [1 2 3 4] * [1; 2; 3; 4]`

   (e) `e = [1 2; 3 4; 5 6] * [1 2 3; 4 5 6]`

   The last two are examples of *matrix multiplication*.

6. Generate an array that contains the values of the function $y = \sin^2(x)$ over the interval $[0, 2\pi]$, with a step size of 0.01. Use the `help` command, determine how the `stem` and `plot` commands work, and plot the function you just generated using both commands. Which is more useful for depicting discrete-time signals, and which is more useful for depicting continuous-time signals?

7. Initialize the array `1:1:10`. Determine and explain the effects of the following commands:

   (a) `array(2)`

   (b) `array(0)`

   (c) `array(11)`

(d) `array(11) = 2;`
`array(11)`
`array`

(e) `array(15) = 8;`
`array`

(f) `array(length(array) + 1) = 9001;`
`array`

3 Creating a MathScript

3.1 Example VI

Oftentimes, we will not want to only run single MathScript commands. We would also like to be able to write whole chunks of commands, edit them as necessary, and then run them as one; this is called a **script**. The following exercise will demonstrate how to create a custom script in a .m file using the MathScript Window.

SCRIPT

1. Open the LabVIEW MathScript Window by going into Tools → MathScript Window from the LabVIEW environment.

2. Write a script that converts units of degrees into radians.

   (a) Click the Script tab to display the Script page.

   (b) Write the following script into the Script Editor on the Script page. Notice that the script includes percent signs (%) that indicate the line is a comment. Use these comments to include descriptive content for the script. Comments immediately after the function name will appear when you use the command help, followed by the name of the function.

   ```
   function deg = rad2deg(rad);
   % This is a comment.
   % This function converts from radians to degrees.
   deg = rad.*180./pi;
   ```

3. Save the script as rad2deg.m.

   (a) Click the Save button to save the function as rad2deg.m in the LabVIEW Data directory. Note that subsequent attempts to save the function can be done via Save in the File menu. By default, LabVIEW searches the LabVIEW Data directory for user-defined functions and user-defined scripts.

   (b) The user-defined function rad2deg is now available as a function you can call from other MathScript scripts.

You can configure the default location where MathScript should search in order to find user-defined functions and user-defined scripts. In the MathScript Window, go to File → LabVIEW MathScript Properties. Change the path, or add another path, under the MathScript: Search Paths tab.

4. In the Command Window, enter the following commands:

   (a) `rad2deg(pi)`
   Invokes the user-defined function you previously defined.

   (b) `help rad2deg`
   Returns the first commented paragraph as the help documentation for the user-defined function.

(c) `rad2deg(linspace(0, pi, 10))`
Invokes the user-defined function that you previously defined with a function call (`linspace`) as a parameter. What does the `linspace` function do?

> Refer to the `Function Syntax` and `Calling User-Defined Functions` sub-topics, under the `MathScript` topic in LabVIEW Help, for more information and help about working with user-defined functions.

3.2 Self-Exercises

1. Determine how the `for` and the `if` commands work, before proceeding with the exercises below. You may find that `help` command useful.

2. Write a script which sums up the elements of an input array and returns that sum. Assume no bound on the length of the array. Save the function as `TotalArray.m`.

   In order to return a value (be it a scalar or an array) from a script, your function will begin with the following line:

   ```
   function total = TotalArray(X);
   ```

   and you will store the final result in `total` in the script. You may *not* use the `sum` function.

   > Note that if you use a `MathScript` function name as a variable later on in your script, the functionality of the function will be clobbered, and invoking the function name will only produce the value of the variable.

   To test your script, enter the following in the `Command Window`:

   ```
   X = [1 2 3 4 5 6 7 8 9 10];
   Y = [1 -1 1 -1 1 -1 1 -1];
   Z = [1];
   XSum = TotalArray(X)
   YSum = TotalArray(Y)
   ZSum = TotalArray(Z)
   ```

   > Note that you can copy and paste directly from this PDF document into `MathScript`.

   This should return three variables with the following content:

   ```
   XSum = 55
   YSum = 0
   ZSum = 1
   ```

3. Write a script which decimates an input array into one array of even indices and another of odd indices. Your script must take in a single array and output two decimated arrays. Save the function as `DecimateArray.m`.

   To test your script, enter the following in the `Command Window`:

   ```
   X = [3 1 4 1 5 9 2 6 5 3];
   [evenArray, oddArray] = DecimateArray(X)
   ```

   This should return two arrays with the following content:

   ```
   evenArray = [1 1 9 6 3]
   oddArray = [3 4 5 2 5]
   ```

   You may find the `mod` function useful. You may also use the following template (you can copy and paste it):

```
function [evenArray, oddArray] = DecimateArray(array);
% This function splits the input array into two
% separate arrays, one containing the elements
% at even indices, another containing the elements
% at odd indices.

% Initialize outputs to empty arrays.
evenArray = [];
oddArray = [];

for % REPLACE THIS COMMENT WITH THE CORRECT ARGUMENTS
    if (% REPLACE THIS COMMENT WITH THE CORRECT ARGUMENT)
        % COMPLETE THE CONSEQUENT CASE OF THE IF-STATEMENT
    else
        % COMPLETE THE ALTERNATIVE CASE OF THE IF-STATEMENT
    end
end
```

4 MathScript Nodes

4.1 Example VI

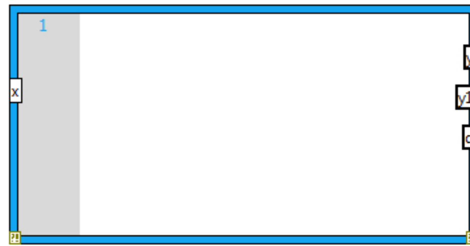
The following exercise will introduce you to using MathScript Nodes in a LabVIEW VI. These MathScript Nodes allow us to integrate MathScript code into the block diagram of a LabVIEW VI, allowing for block diagrams to become more compact and powerful.

1. Create a new LabVIEW VI called MathScript Node.vi.
2. Place a MathScript Node on the block diagram.
   (a) Right-click on the diagram and navigate to Programming → Structures → MathScript Node. Click and drag the mouse pointer to place a MathScript Node on the block diagram.
3. Add input and output terminals to the MathScript Node.
   (a) Right-click on the left border of the MathScript Node and select Add Input. Name the input x. Naming is performed by double clicking on the box that appears and typing in the desired name.
   (b) Right-click on the right border and select Add Output. Repeat this process to create three outputs labeled y, y1, and d.
4. In the MathScript Node, type the following expressions:

   ```
   y = x.^2;
   y1 = y(1);
   ```

   After this step, your block diagram should match that as shown in Figure 2. The labels of the terminals are black, because the terminals do not have data types assigned to them yet.
5. Create a control for the input terminal. On the front panel, place an array of numeric controls. Name the array x and wire it to the x input of the MathScript Node in the block diagram. In general, however, the name of the control in the block diagram does not have to match the name of the MathScript Node input.

Figure 2 `MathScript Node` with Terminals.



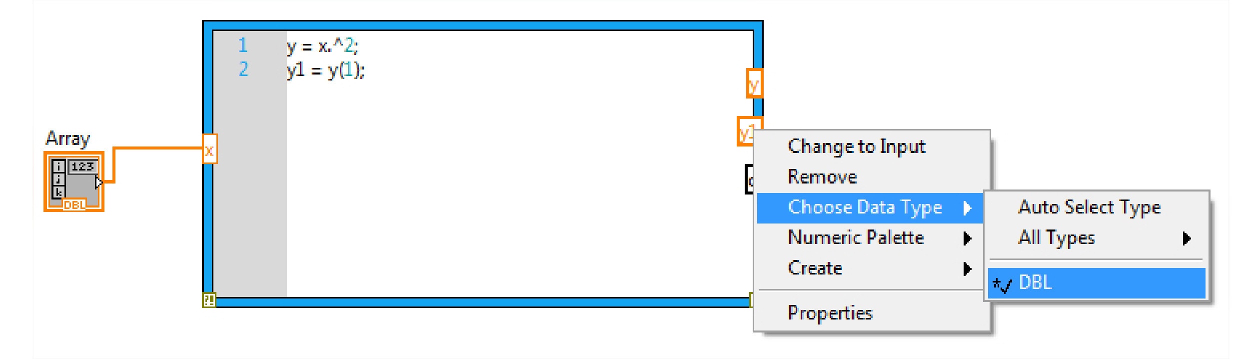
4.2 Data Types of Output Terminals

Notice that, because the `x` input of the `MathScript Node` is connected to an array of doubles, the label is now orange, the same as the color of the wire feeding into the terminal.

More importantly, notice that the `MathScript Node` has predicted that the data type of `y` will be an array of doubles, and that the data type of `y1` will be a scalar double; as a result, it has made the labels of these output terminals orange as well. To confirm this, right-click on the terminal and select `Choose Data Type`. As shown in Figure 3, the current data type of the terminal will be preceded by an asterisk to denote that the data type was automatically determined.

In general, `MathScript Node`s assign an output terminal with the smallest data type that can hold the possible output values of the terminal. You can, of course, choose a different data type if the predicted data type is not the one required. **This is another common cause of error in LabVIEW VIs** and will definitely happen to you at least once: when specifying the outputs of a `MathScript Node`, *ensure that each of them are of the correct data type*. This will save you a lot of time and frustration. A good debugging tip is to **notice the thickness of wires**: as we learned in lab 02, a wire carrying an array is thicker than a wire carrying a scalar.

Figure 3 Checking and Selecting Data Types for `MathScript Node` Terminals.



6. The output terminal `d` still has a black label, since it has not yet been used inside the `MathScript Node`. Append the following line to the code inside the `MathScript Node`:

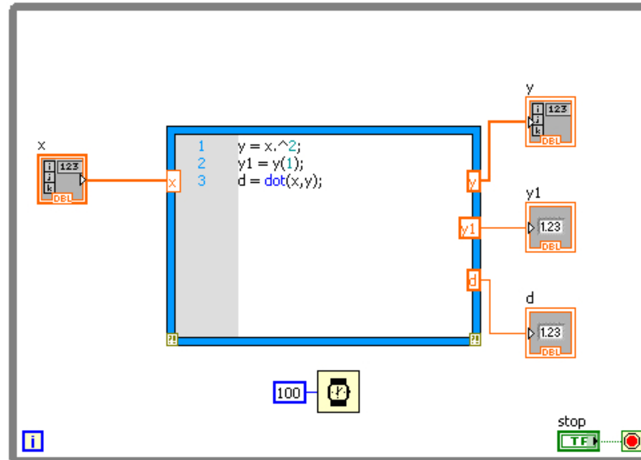
   ```
   d = dot(x, y);
   ```

   Click outside the `MathScript Node` to refresh the data type of `d`. What is its data type now?

7. Place a `While Loop` with a `Stop` button around the `MathScript Node` and the controls. Be sure to include a `Wait (ms)` function inside the loop to conserve memory usage.

8. Create indicators for each of the outputs `y`, `y1`, and `d`. Determine the data types for each of the outputs by the commands in the `Mathscript Node`, and place indicators for these outputs on the front panel. Next, wire each `Mathscript Node` output to the appropriate indicator. Be sure to right-click on each `Mathscript Node` output to make sure that the correct data type is selected. Your block diagram should resemble Figure 4.

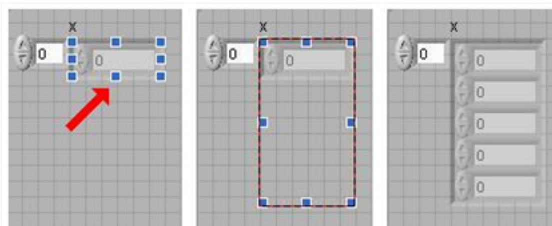
Figure 4 `MathScript Node.vi` Block Diagram.



You can clean up the wires in your block diagram, by right-clicking on a wire and selecting `Clean Up Wire`. Alternatively, the key combination `Ctrl-U` cleans up *all* wires.

9. On the front panel, expand the arrays to show multiple elements. With the cursor, grab the bottom middle selector of the array and drag it down to show multiple elements.

Figure 5 `x` Input Array Control on the Front Panel.

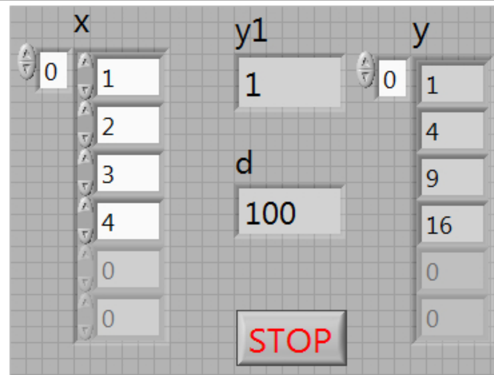


10. Begin by placing a `1`, `2`, `3`, and `4` in the first four elements of the `x` control. Your front panel should look similar to the one in Figure 6.

    Note that the later elements in the array are grayed out. This is because they are not initialized. You can initialize them by clicking inside the cell and entering a value. To uninitialize a cell, right-click on the element and select `Data Operations → Delete Element`.

11. Run the VI. Change the values of the elements in the array to see how the outputs change.

Figure 6 `MathScript Node.vi` Front Panel.



4.3 Self-Exercise

Write a script which implements `Previous Element Sum.vi` from lab 02 using a `MathScript Node`. If done correctly, you will not need to use a `For Loop` *structure*, although the `for` *command* will prove very useful. Save the VI as `Previous Element Sum MathScript.vi`. Test your new VI and verify that it functions identically to `Previous Element Sum.vi`.

5 Miscellaneous Exercises

Now that we have covered the basic concepts and structures of LabVIEW, we are now ready to tackle a few miscellaneous exercises. The objective of the exercises that follow is to determine which LabVIEW constructs from the past three labs will be best suited for the job, and how they are to be used, given the restrictions provided.

1. Implement a VI that separates the even and odd indexed elements of an array; a representative figure of the solution is shown in Figure 7. This is similar to the exercise presented in section 3. Save this VI as `Array Decimation.vi`.

   You have the following restrictions:

   (a) Do *not* use the `Decimate 1D Array` block.

   (b) Do *not* use `MathScript Nodes`.

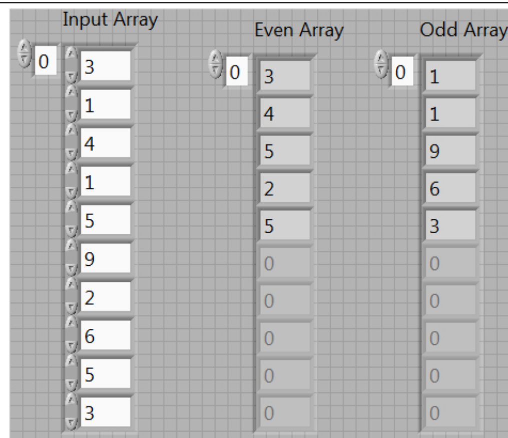
   You may find the following hints useful:

   (a) Consider using the `Build Array` block and case structures.

   (b) Shift registers *can* carry arrays as well, besides constants. In this case, the initializers for the shift registers will not be constants, but will be empty arrays.

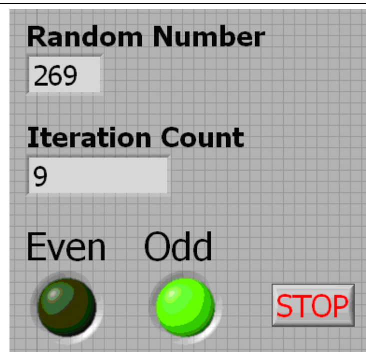
   (c) If you are still stuck, consider a similar problem, that of the `Random Competition` VI from lab 02. In that case, you added `1` to either competitor A or competitor B based on a condition. The process is similar here, only this time you will be appending array elements to either the `Even Array` or the `Odd Array` based on a particular condition.

Figure 7 `Array Decimation.vi` Front Panel.



2. Implement a VI that generates a random number between `0` and `1000` every 0.5 second and determines whether this number is odd or even. This VI should display the random number that is generated, the current iteration count, and two boolean indicators that declare the odd-even state of the number. A representative figure of the solution is shown in Figure 8. Save this VI as `Odd Even.vi`.

Figure 8 `Odd Even.vi` Front Panel.



6 Acknowledgments

Special thanks go out to the teaching assistants (TAs) of the Spring 2009 semester (Vinay Raj Hampapur, Miklos Christine, Sarah Wodin-Schwartz), of the Fall 2009 semester (David Carlton, Judy Hoffman, Mark Landry, Feng Pan, Changho Suh), and of the Spring 2010 semester (Xuan Fan, Brian Lambson, Kelvin So) for providing suggestions, ideas, and fixes to this lab guide. Portions of this lab guide were derived from material contained in the LabVIEW reference guides published by National Instruments.