The Benefits of Programming Graphically in NI LabVIEW

For more than 20 years, NI LabVIEW has been used by millions of engineers and scientists to develop sophisticated test, measurement, and control applications. While LabVIEW provides a variety of features and tools ranging from interactive assistants to configurable user-defined interfaces, it is differentiated by its graphical, general-purpose programming language (known as G) along with an associated integrated compiler, a linker, and debugging tools.

A Brief History of the Pursuit of Higher-Level Programming

To better understand the major value propositions of LabVIEW graphical programming, it is helpful to review some background on the first higher-level programming language. At the dawn of the modern computer age in the mid-1950s, a small team at IBM decided to create a more practical alternative to programming the enormous IBM 704 mainframe (a supercomputer in its day) in low-level assembly language, the most modern language available at the time. The result was FORTRAN, a more human-readable programming language whose purpose was to speed up the development process.

The engineering community was initially skeptical that this new method could outperform programs hand-crafted in assembly, but soon it was shown that FORTRAN-generated programs ran nearly as efficiently as those written in assembly. At the same time, FORTRAN reduced the number of programming statements necessary in a program by a factor of 20, which is why it is often considered the first higher-level programming language. Not surprisingly, FORTRAN quickly gained acceptance in the scientific community and remains influential.

Fifty years later, there are still important lessons in this anecdote. First, for more than 50 years, engineers have sought easier and faster ways to solve problems through computer programming. Second, the programming languages chosen by engineers to translate their tasks have trended toward higher levels of abstraction. These lessons help explain the immense popularity and widespread adoption of G since its inception in 1986; G represents an extremely high-level programming language whose purpose is to increase the productivity of its users while executing at nearly the same speeds as lower-level languages like FORTRAN, C, and C++.

LabVIEW: Graphical, Dataflow Programming

LabVIEW is different from most other general-purpose programming languages in two major ways. First, G programming is performed by wiring together graphical icons on a diagram, which is then compiled directly to machine code so the computer processors can execute it.

While represented graphically instead of with text, G contains the same programming concepts found in most traditional languages. For example, G includes all the standard constructs, such as data types, loops, event handling, variables, recursion, and object-oriented programming.

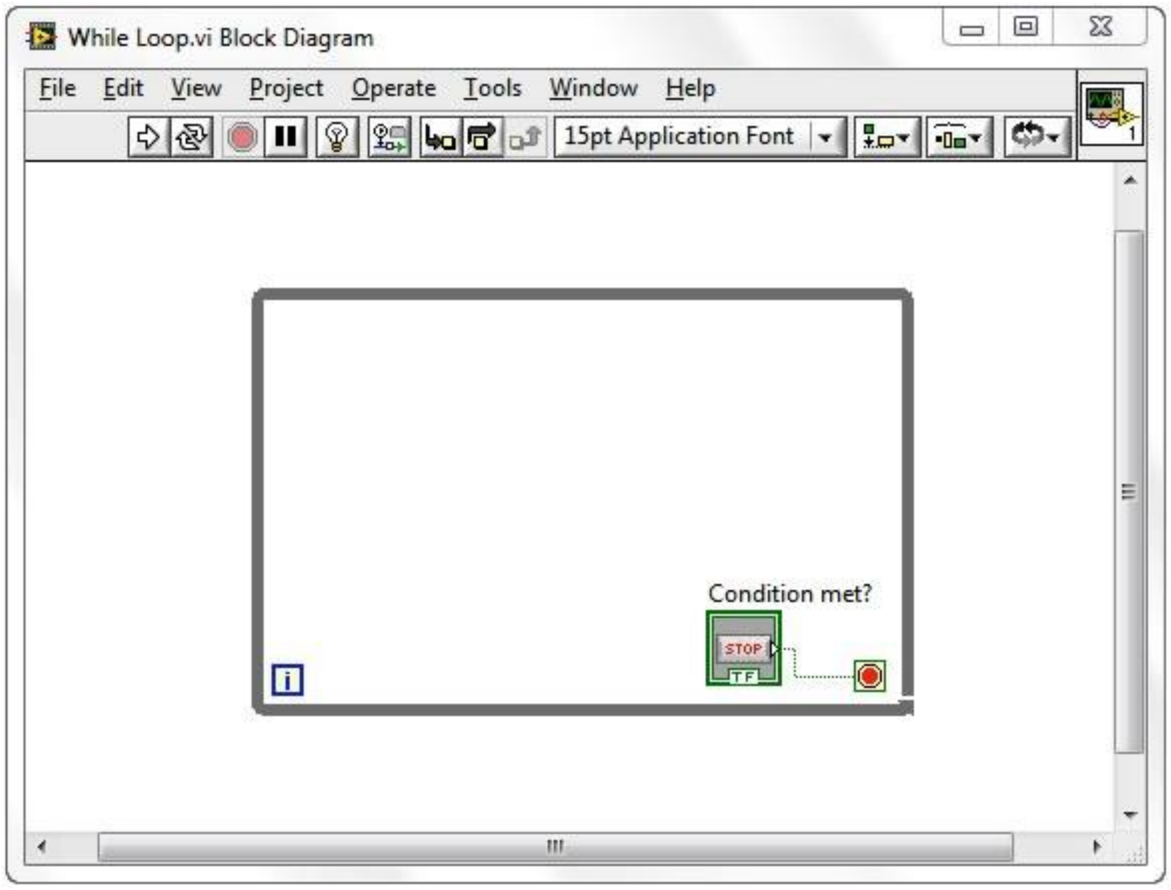


Figure 1. A While Loop in G is intuitively represented by a graphical loop, which executes until a stop condition is met.

The second main differentiator is that G code developed with LabVIEW executes according to the rules of data flow instead of the more traditional procedural approach (in other words, a sequential series of commands to be carried out) found in most text-based programming languages like C and C++. Dataflow languages like G (as well as Agilent VEE, Microsoft Visual Programming Language, and Apple Quartz Composer) promote data as the main concept behind any program. Dataflow execution is data-driven, or data-dependent. The flow of data between nodes in the program, not sequential lines of text, determines the execution order.

This distinction may seem minor at first, but the impact is extraordinary because it renders the data paths between parts of the program to be the developer's main focus. Nodes in a LabVIEW program (in other words, functions, structures such as loops, subroutines, and so on) have inputs, process data, and produce outputs. Once all of a given node's inputs contain valid data, that node executes its logic, produces output data, and passes that data to the next node in the dataflow path. A node that receives data from another node can execute only after the other node completes execution.

Benefits of G Programming

Intuitive Graphical Programming

Like most people, engineers and scientists learn by seeing and processing images without any need for conscious contemplation. Many engineers and scientists can also be characterized as "visual thinkers," meaning that they are especially adept at using visual processing to organize information. In other words, they think best in pictures. This is often reinforced in colleges and universities, where students are encouraged to model solutions to problems as process diagrams. However, most general-purpose programming languages require you to spend significant time learning the specific text-based syntax associated with that language and then map the structure of the language to the problem being solved. Graphical programming with G provides a more intuitive experience.

G code is typically easier for engineers and scientists to quickly understand because they are largely familiar with visualizing and even diagrammatically modeling processes and tasks in terms of block diagrams and flowcharts (which also follow the rules of data flow). In addition, because dataflow languages require you to base the structure of the program around the flow of data, you are encouraged to think in terms of the problem you need to solve. For example, a typical G program might first acquire several channels of temperature data, then pass the data to an analysis function, and, finally, write the analyzed data to disk. Overall, the flow of data and steps involved in this program are easy to understand within a LabVIEW diagram.

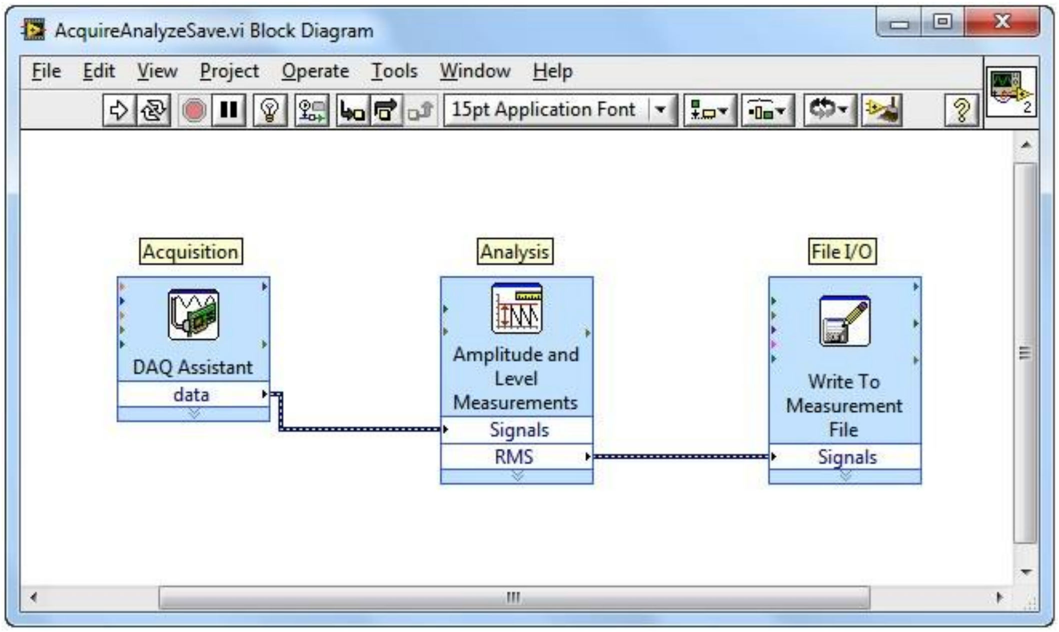

AcquireAnalyzeSave.vi Block Diagram
File Edit View Project Operate Tools Window Help
15pt Application Font
Acquisition
DAQ Assistant
data
Analysis
Amplitude and Level Measurements
Signals
RMS
File I/O
Write To Measurement File
Signals

Figure 2. Data originates in the acquisition function and then flows intuitively to the analysis and storage functions through wires.

Interactive Debugging Tools

Because LabVIEW graphical G code is easy to comprehend, common programming tasks, like debugging, become more intuitive as well. For example, LabVIEW provides unique debugging tools that you can use to watch as data interactively moves through the wires of a LabVIEW program and see the data values as they pass from one function to another along the wires (known within LabVIEW as execution highlighting).

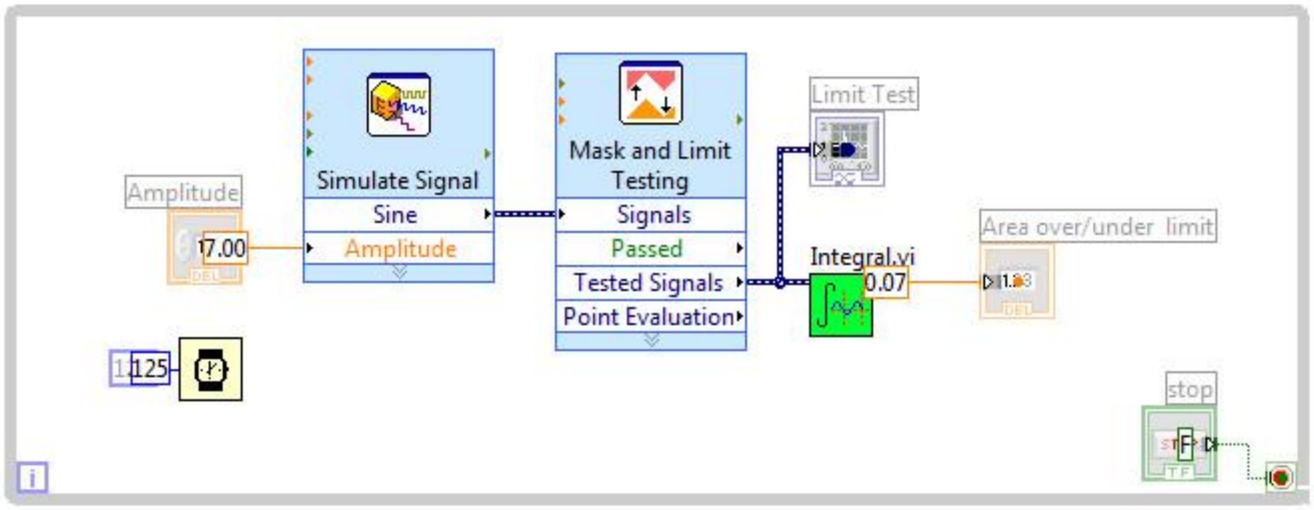


Figure 3. Highlight execution provides an intuitive way to understand the execution order of G code.

LabVIEW also offers debugging features for G comparable to those found in traditional programming tools. These features, accessible as part of the toolbar for a diagram, include probes, breakpoints, and step over/into/out of.

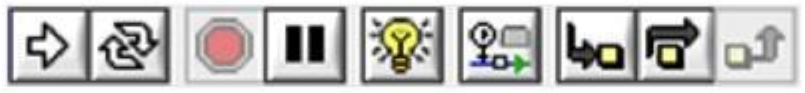


Figure 4. The block diagram toolbar offers access to standard debugging tools like stepping.

With the G debugging tools, you can probe data on many parts of the program simultaneously, pause execution, and step into a subroutine without complex programming. While this is possible in other programming languages, it is easier to visualize the state of the program and the relationships between parallel parts of the code (which are common in G because of its graphical nature).

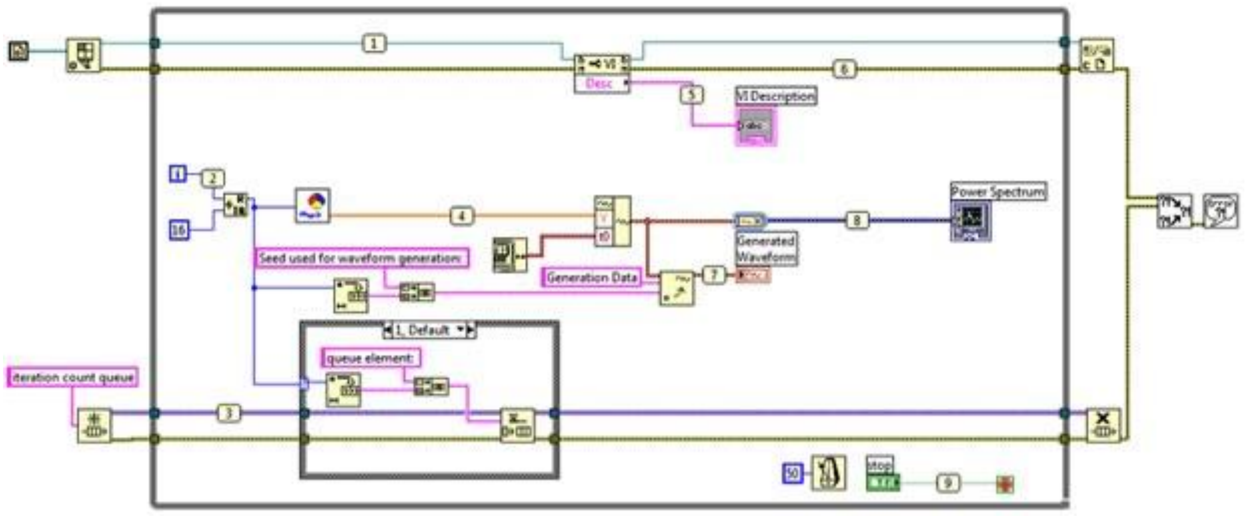
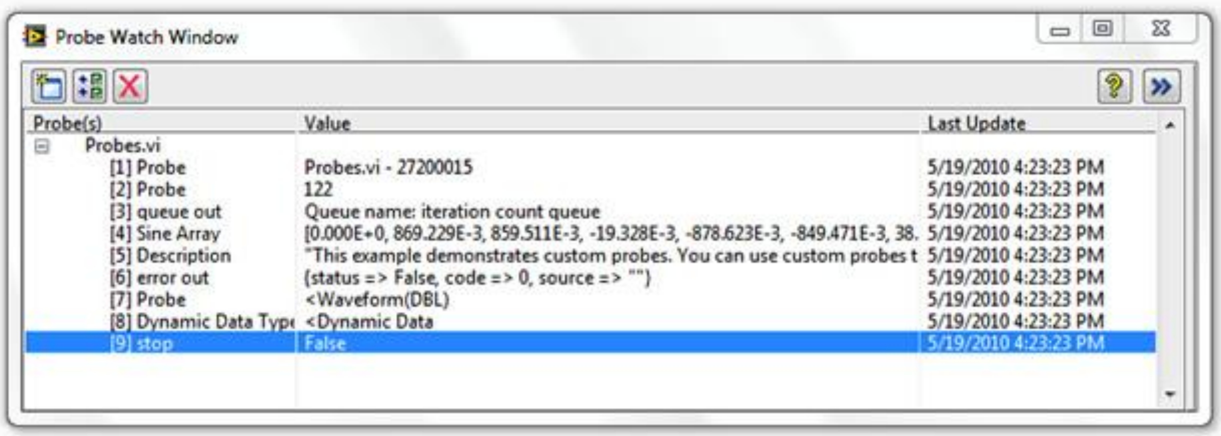


Figure 5. Probes are effective ways in LabVIEW to see values traveling on wires throughout the application, even for parallel sections of code.



Probe(s)	Value	Last Update
Probes.vi		
[1] Probe	Probes.vi - 27200015	5/19/2010 4:23:23 PM
[2] Probe	122	5/19/2010 4:23:23 PM
[3] queue out	Queue name: iteration count queue	5/19/2010 4:23:23 PM
[4] Sine Array	[0.000E+0, 869.229E-3, 859.511E-3, -19.328E-3, -878.623E-3, -849.471E-3, 38.	5/19/2010 4:23:23 PM
[5] Description	"This example demonstrates custom probes. You can use custom probes t	5/19/2010 4:23:23 PM
[6] error out	(status => False, code => 0, source => "")	5/19/2010 4:23:23 PM
[7] Probe	<Waveform(DBL)>	5/19/2010 4:23:23 PM
[8] Dynamic Data Type	<Dynamic Data	5/19/2010 4:23:23 PM
[9] stop	False	5/19/2010 4:23:23 PM

Figure 6. View probe values in the Probe Watch Window, which displays probe values for any probes in the entire application (including subroutines).

One of the most common debugging features used in LabVIEW is the always-on compiler. While you are developing a program, the compiler continuously checks for errors and provides semantic and syntactic feedback on the application. If an error exists, you cannot run the program – you see only a broken Run button in the toolbar.

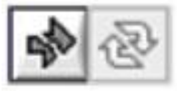
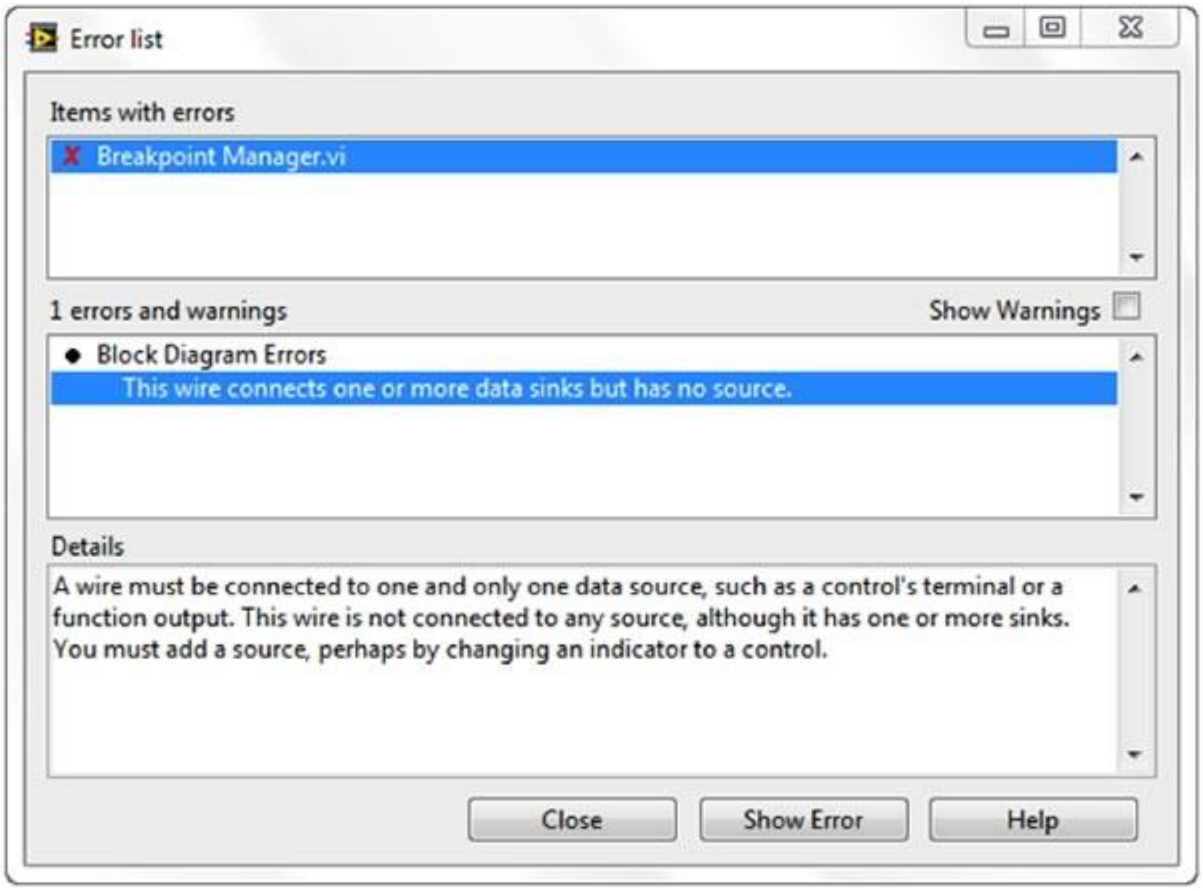


Figure 7. The broken Run arrow provides immediate feedback indicating syntactical errors in the G code.

Pressing the broken Run button opens a list of problems that you must address. Once you have addressed these issues the LabVIEW compiler can compile your program to machine code. Once compiled, the performance of G programs is comparable to that of more traditional text-based languages like C.



This wire connects one or more data sinks but has no source.

Details

A wire must be connected to one and only one data source, such as a control's terminal or a function output. This wire is not connected to any source, although it has one or more sinks. You must add a source, perhaps by changing an indicator to a control.

Figure 8. The error list displays a detailed explanation of each syntax error in the entire code hierarchy.

Automatic Parallelism and Performance

Dataflow languages like LabVIEW allow for automatic parallelization. In contrast to sequential languages like C and C++, graphical programs inherently contain information about which parts of the code should execute in parallel. For example, a common G design pattern is the

Producer/Consumer Design Pattern, in which two separate While Loops execute independently: the first loop is responsible for producing data and the second loop processes data. Despite executing in parallel (possibly at different rates), data is passed between the two loops using queues, which are standard data structures in general-purpose programming languages.

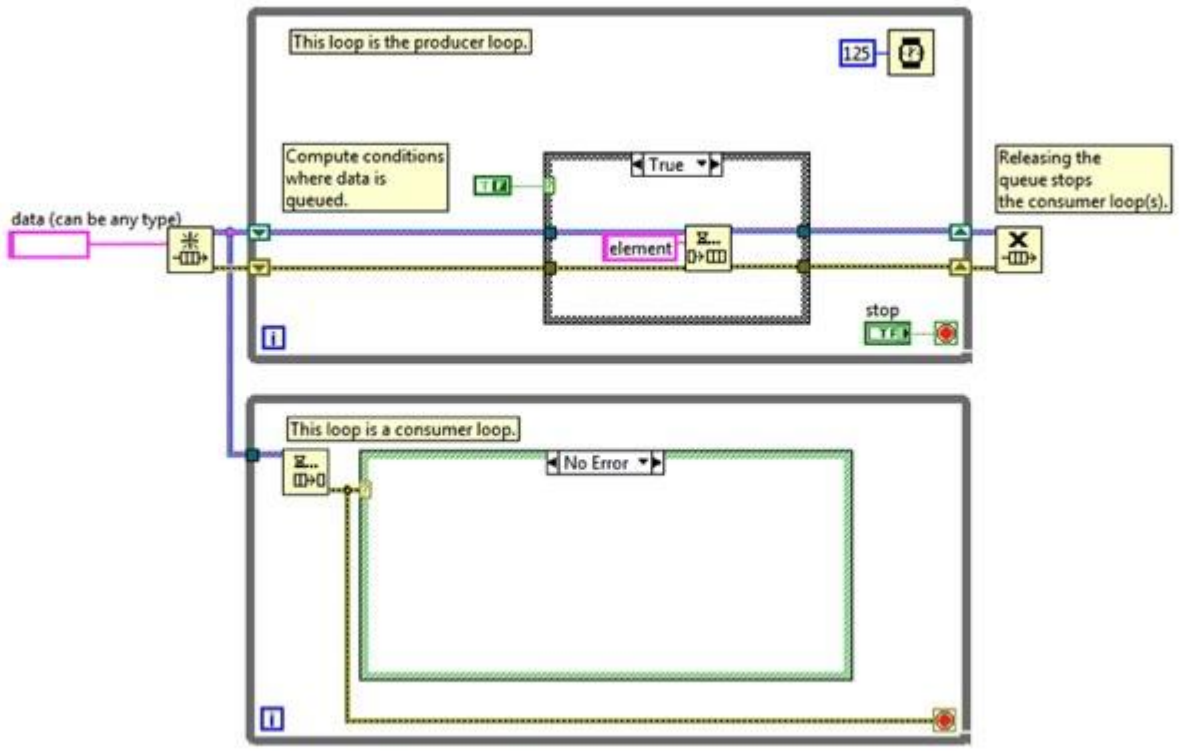

This loop is the producer loop.
125
Compute conditions where data is queued.
True
data (can be any type)
element
Releasing the queue stops the consumer loop(s).
stop
This loop is a consumer loop.
No Error

Figure 9. The LabVIEW Producer/Consumer design pattern is often used to increase the performance of applications that require parallel tasks.

Parallelism is important in computer programs because it can unlock performance gains relative to purely sequential programs due to recent changes in computer processor designs. For more than 40 years, computer chip manufacturers increased processor clock speed to increase chip performance. Today, however, increasing clock speeds for performance gains is no longer viable because of power consumption and heat dissipation constraints. As a result, chip vendors have instead moved to new chip architectures with multiple processor cores on a single chip.

To take advantage of the performance available in multicore processors, you must be able to use multithreading within your applications (in other words, break up applications into discrete sections that can be executed independently). If you use traditional text-based languages, you must explicitly create and manage threads to implement parallelism, a major challenge for nonexpert programmers.

In contrast, the parallel nature of G code makes multitasking and multithreading simple to implement. The built-in compiler continually works in the background to identify parallel sections of code. Whenever G code has a branch in a wire, or a parallel sequence of nodes on the diagram, the compiler tries to execute the code in parallel within a set of threads that LabVIEW manages automatically. In computer science terms, this is called "implicit parallelism" because you do not have to specifically write code with the purpose of running it in parallel; the G language takes care of parallelism on its own.

Beyond multithreading on a multicore system, G can provide even greater parallel execution by extending graphical programming to field-programmable gate arrays (FPGAs). FPGAs are reprogrammable silicon chips that are massively parallel – with each independent processing task assigned to a dedicated section of the chip – but they are not limited by the number of processing cores available. As a result, the performance of one part of the application is not adversely affected when more processing is added.

Historically, FPGA programming was the province of only a specially trained expert with a deep understanding of digital hardware design languages. Increasingly, engineers without FPGA expertise want to use FPGA-based custom hardware for unique timing and triggering routines, ultrahigh-speed control, interfacing to digital protocols, digital signal processing (DSP), RF and communications, and many other applications requiring high-speed hardware reliability, customization, and tight determinism. G is particularly suited for FPGA programming because it clearly represents parallelism and data flow and is quickly growing in popularity as a tool of choice for developers seeking parallel processing and deterministic execution.

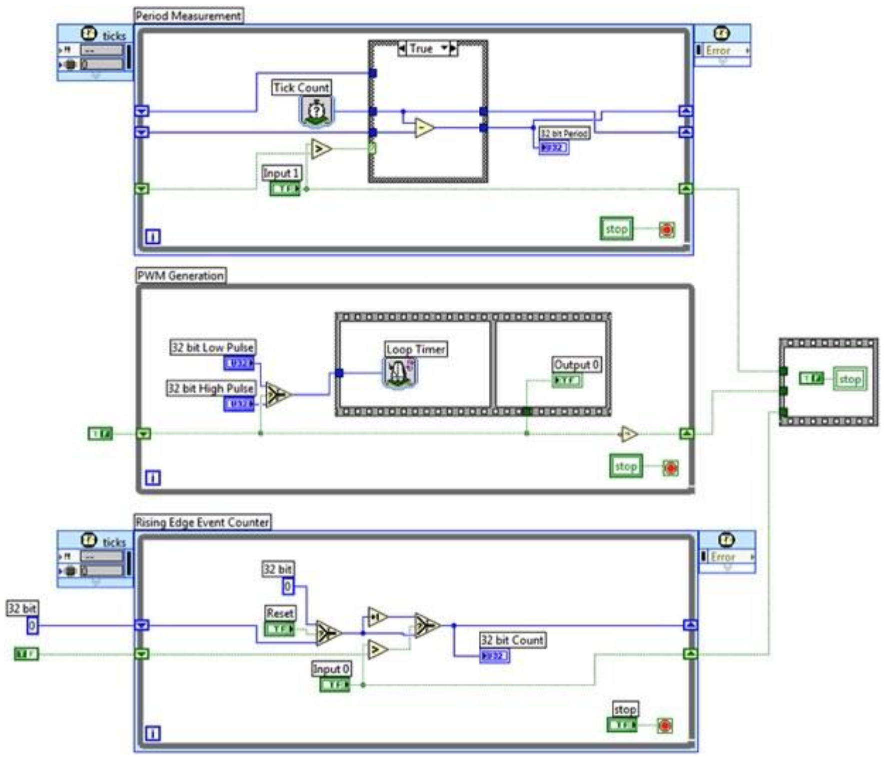


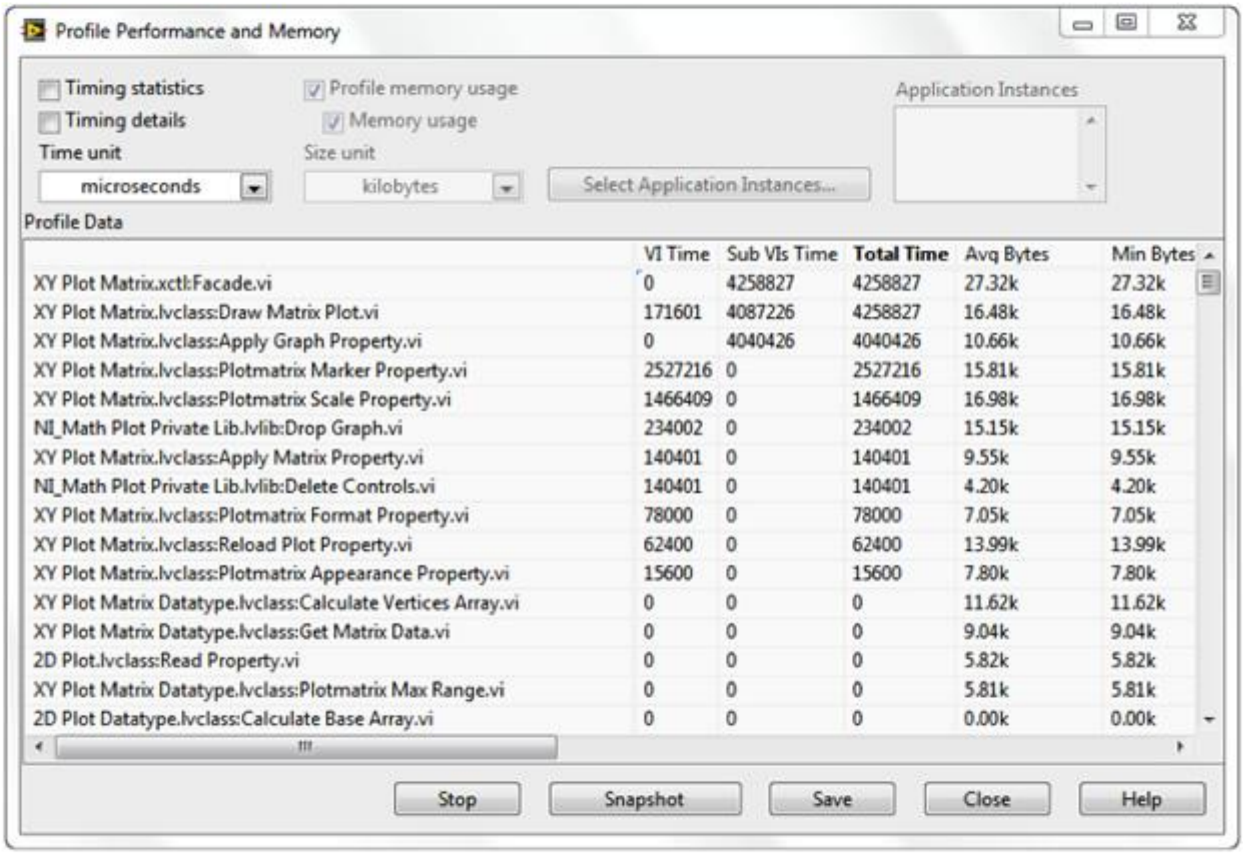
Figure 10. LabVIEW FPGA code with parallelism becomes truly independent pathways on the FPGA silicon.

Abstraction of Low-Level Tasks

As demonstrated in the FORTRAN narrative, abstraction is one of the principal benefits of higher-level programming languages, which express programs in more intuitive ways when compared with lower-level languages. G automatically handles many of the challenges that you normally must contend with in a text-based programming language (like memory usage). In a text-based language, you are responsible for allocating memory before using it and de-allocating it once it is no longer needed. You also must remember to be careful not to write past the end of the memory allocated in the first place. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes you can make in text-based languages. Inadequate memory allocation is also a difficult problem to debug.

Automatic memory handling is one of the chief benefits of programming in G. Using G, you do not allocate variables nor assign values to and from them. Instead, as explained earlier, you create a diagram with connections representing the transition of data. Nodes on a LabVIEW diagram that generate data automatically handle allocating the storage for that data. When data is no longer being used, the associated memory is also de-allocated automatically. If you add new information to an array or a string, more memory is automatically allocated to manage the new information. The removal of the low-level memory management details frees you to focus on the problem you are trying to solve instead of studying complex rules associated with preventing a run-time error in the execution of the program.

At the same time, if you want lower-level control of G memory usage, you can use built-in memory management tools to help monitor memory; this is part of an opt-in approach that you can decide to use in a particular application or even a portion of an application. If you determine that memory usage is an issue for a LabVIEW program, you can intervene to lower the amount of memory you are using through more advanced programming techniques.


Profile Performance and Memory
Select Application Instances...
Stop
Snapshot
Save
Close
Help

Application Instances

[ ] Timing statistics
[ ] Timing details
[✓] Profile memory usage
[✓] Memory usage
Time unit: microseconds
Size unit: kilobytes

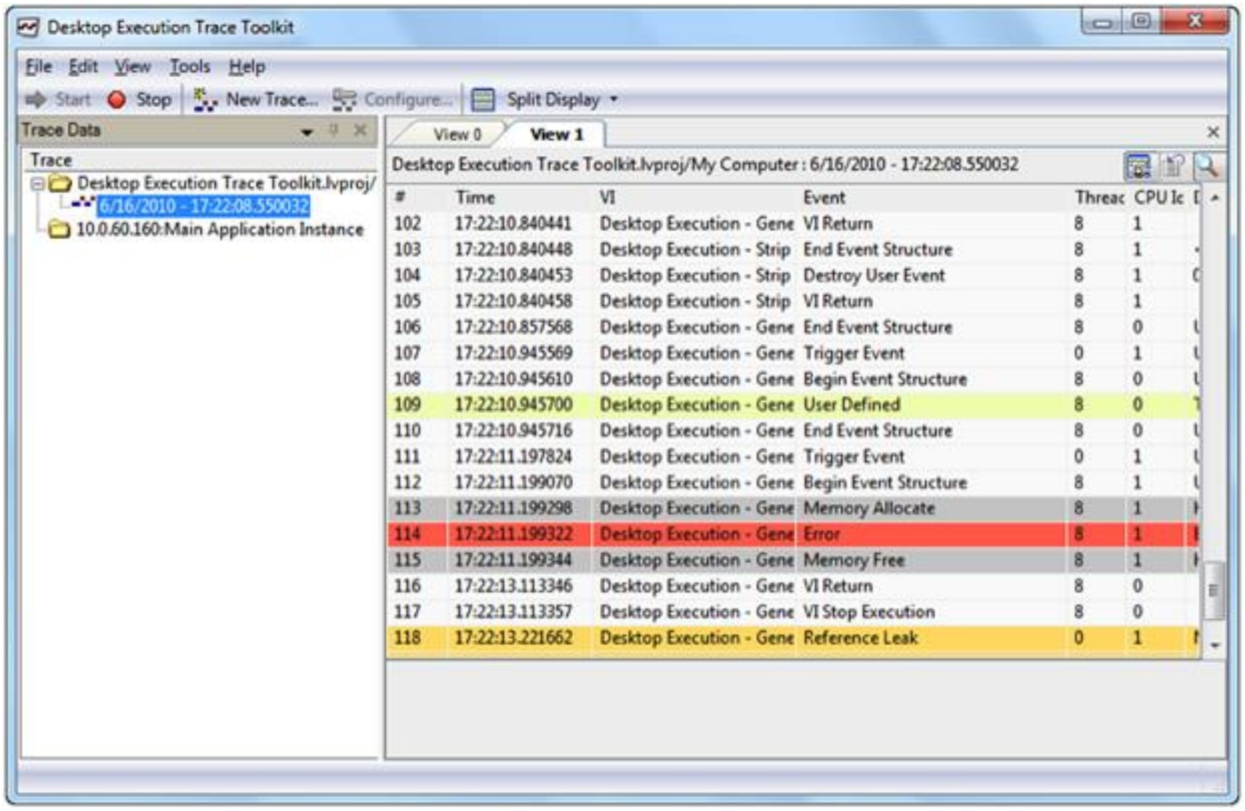
Profile Data

	VI Time	Sub VIs Time	Total Time	Avg Bytes	Min Bytes
XY Plot Matrix.xctl:Facade.vi	0	4258827	4258827	27.32k	27.32k
XY Plot Matrix.lvclass:Draw Matrix Plot.vi	171601	4087226	4258827	16.48k	16.48k
XY Plot Matrix.lvclass:Apply Graph Property.vi	0	4040426	4040426	10.66k	10.66k
XY Plot Matrix.lvclass:Plotmatrix Marker Property.vi	2527216	0	2527216	15.81k	15.81k
XY Plot Matrix.lvclass:Plotmatrix Scale Property.vi	1466409	0	1466409	16.98k	16.98k
NI_Math Plot Private Lib.lvlib:Drop Graph.vi	234002	0	234002	15.15k	15.15k
XY Plot Matrix.lvclass:Apply Matrix Property.vi	140401	0	140401	9.55k	9.55k
NI_Math Plot Private Lib.lvlib:Delete Controls.vi	140401	0	140401	4.20k	4.20k
XY Plot Matrix.lvclass:Plotmatrix Format Property.vi	78000	0	78000	7.05k	7.05k
XY Plot Matrix.lvclass:Reload Plot Property.vi	62400	0	62400	13.99k	13.99k
XY Plot Matrix.lvclass:Plotmatrix Appearance Property.vi	15600	0	15600	7.80k	7.80k
XY Plot Matrix Datatype.lvclass:Calculate Vertices Array.vi	0	0	0	11.62k	11.62k
XY Plot Matrix Datatype.lvclass:Get Matrix Data.vi	0	0	0	9.04k	9.04k
2D Plot.lvclass:Read Property.vi	0	0	0	5.82k	5.82k
XY Plot Matrix Datatype.lvclass:Plotmatrix Max Range.vi	0	0	0	5.81k	5.81k
2D Plot Datatype.lvclass:Calculate Base Array.vi	0	0	0	0.00k	0.00k

Figure 11. Managing memory in LabVIEW is optional, but advanced users can profile memory usage to help identify portions of the application to optimize.

When G code exhibits unusual or unexpected behavior that you cannot easily resolve using the previously mentioned debugging tools, you can use more advanced debugging features with the LabVIEW Desktop Execution Trace Toolkit. This toolkit is designed for more advanced users who want to perform dynamic code analysis for the following:

- Detecting memory and reference leaks
- Isolating the source of a specific event or undesired behavior
- Screening applications for areas where performance can be improved
- Identifying the last call before an error
- Ensuring the execution of an application is the same on different targets


Desktop Execution Trace Toolkit
File Edit View Tools Help
Start Stop New Trace... Configure... Split Display
Trace Data
Trace
Desktop Execution Trace Toolkit.lvproj/
6/16/2010 - 17:22:08.550032
10.0.60.160:Main Application Instance
View 0
View 1

Desktop Execution Trace Toolkit.lvproj/My Computer : 6/16/2010 - 17:22:08.550032

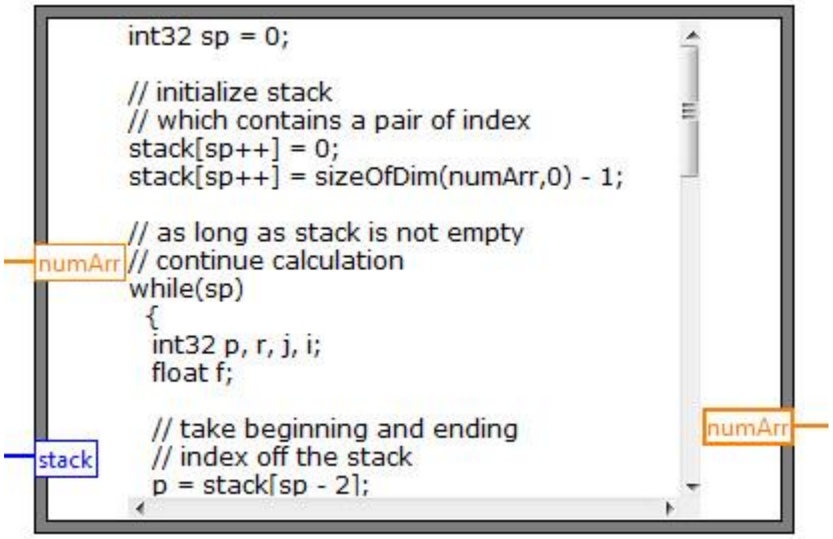
#	Time	VI	Event	Threac	CPU Ic
102	17:22:10.840441	Desktop Execution - Gene	VI Return	8	1
103	17:22:10.840448	Desktop Execution - Strip	End Event Structure	8	1
104	17:22:10.840453	Desktop Execution - Strip	Destroy User Event	8	1
105	17:22:10.840458	Desktop Execution - Strip	VI Return	8	1
106	17:22:10.857568	Desktop Execution - Gene	End Event Structure	8	0
107	17:22:10.945569	Desktop Execution - Gene	Trigger Event	0	1
108	17:22:10.945610	Desktop Execution - Gene	Begin Event Structure	8	0
109	17:22:10.945700	Desktop Execution - Gene	User Defined	8	0
110	17:22:10.945716	Desktop Execution - Gene	End Event Structure	8	0
111	17:22:11.197824	Desktop Execution - Gene	Trigger Event	0	1
112	17:22:11.199070	Desktop Execution - Gene	Begin Event Structure	8	1
113	17:22:11.199298	Desktop Execution - Gene	Memory Allocate	8	1
114	17:22:11.199322	Desktop Execution - Gene	Error	8	1
115	17:22:11.199344	Desktop Execution - Gene	Memory Free	8	1
116	17:22:13.113346	Desktop Execution - Gene	VI Return	8	0
117	17:22:13.113357	Desktop Execution - Gene	VI Stop Execution	8	0
118	17:22:13.221662	Desktop Execution - Gene	Reference Leak	0	1

Figure 12. If you need lower-level debugging, you can use the LabVIEW Desktop Execution Trace Toolkit to view the dynamic execution characteristics of an application.

Combining G with Other Languages

While G code provides an excellent representation for parallelism and removes the requirement on developers to understand and manage computer memory, it is not necessarily ideal for every task. In particular, mathematical formulas and equations can often be more succinctly represented with text. For that reason, you can use LabVIEW to combine graphical programming with several forms of text-based programming. Working within LabVIEW, you can choose a textual approach, a graphical approach, or a combination of the two.

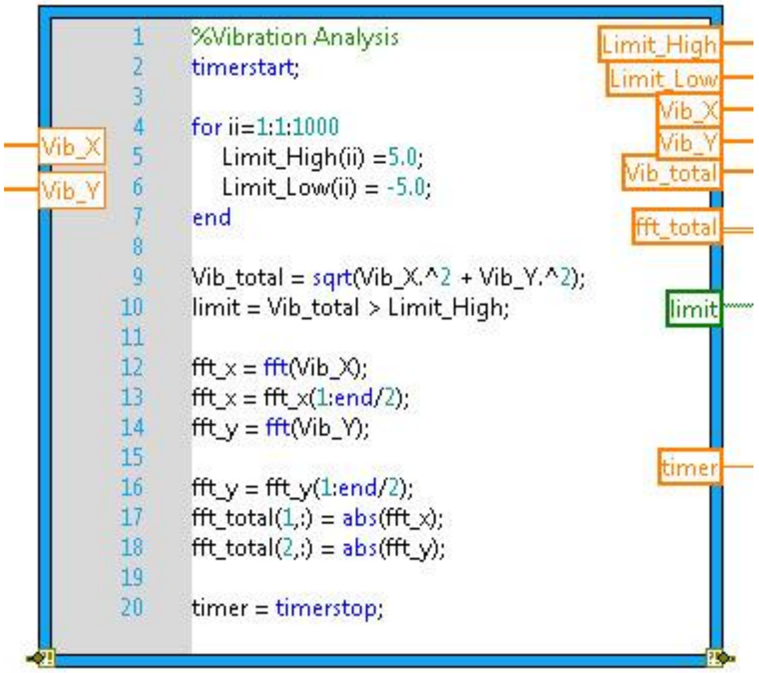
For example, LabVIEW contains the concept of the Formula Node, which evaluates textual mathematical formulas and expressions similar to C on the block diagram. These mathematical formulas can execute side by side and integrate with graphical LabVIEW code.



```
int32 sp = 0;

// initialize stack
// which contains a pair of index
stack[sp++] = 0;
stack[sp++] = sizeOfDim(numArr,0) - 1;

// as long as stack is not empty
// continue calculation
while(sp)
  {
   int32 p, r, j, i;
   float f;

   // take beginning and ending
   // index off the stack
   p = stack[sp - 2];
```

Figure 13. The Formula Node uses syntax similar to C to represent mathematical expressions in a succinct, text-based format.

Similarly, the MathScript Node adds math-oriented, textual programming to LabVIEW that is generally compatible with the commonly used .m file syntax.



```
%Vibration Analysis
timerstart;

for ii=1:1:1000
    Limit_High(ii) =5.0;
    Limit_Low(ii) = -5.0;
end

Vib_total = sqrt(Vib_X.^2 + Vib_Y.^2);
limit = Vib_total > Limit_High;

fft_x = fft(Vib_X);
fft_x = fft_x(1:end/2);
fft_y = fft(Vib_Y);

fft_y = fft_y(1:end/2);
fft_total(1,:) = abs(fft_x);
fft_total(2,:) = abs(fft_y);

timer = timerstop;
```

Figure 14. With the MathScript Node, you can create or reuse .m file scripts for signal processing and data analysis.

A Better Way for You to Solve Problems

LabVIEW and its graphical, dataflow programming language provides a better way for you to solve problems than traditional, lower-level alternatives, and the proof is in its longevity. The key differentiators for programming in G are the intuitive graphical code that you can create and the data-driven rules that govern its execution combine to offer a programming experience that expresses the thought processes of its users more closely than other languages. Despite G being a higher-level language, you can still achieve performance comparable to that of languages like C because of the built-in LabVIEW compiler.