An Introduction to Floating-Point Behavior in LabVIEW

Numerical Implications of Compiler Improvements in LabVIEW 2010

More sensors. More data. More computations. This is a reality faced by scientists and engineers today. The pressure is on to deliver better performance on the latest processor technologies, especially multi-core CPUs. To meet the challenge, LabVIEW 2010 incorporates improvements at the language and compiler levels.

Standing still is not an option. The decline in core frequencies means that existing applications built using yesterday's compiler technologies run slower on today's processors. To achieve expected gains, higher level code must utilize multiple cores and maximize use of each core's resources. Following this path changes the computational landscape of applications.

Since the late 1980s, the floating-point system defined by the IEEE Standard for Binary Floating-Point Arithmetic (IEEE754) has dominated the computer industry. For a major processor, programming language or numerical library to be recognized, it must support most if not all aspects of IEEE754. LabVIEW relies on IEEE754-compliant software and hardware for a significant portion of its numeric functionality.

While IEEE754 offers a solid foundation for porting floating-point implementations, it establishes no explicit tools or guidance for analyzing the error produced by compliant implementations. Fundamental issues tied to floating-point accuracy are not yet addressed by any standards or industry body.

At least one reason for the lack of action is due to a prevailing misconception:

> *The computer industry, by and large, has computational mathematics well in hand. Standards and methods are in place so that most implementations are capable of producing numerical results with sufficient accuracy for many (if not most) applications.*

In the absence of such information, computer solutions remain resilient because the error in most computations doesn't matter. Perhaps the noise in the acquired data overwhelms the contribution of error from a single computation or the amount of time allotted to perform computations avoids significant accumulation of numerical error. Whatever the reason, prior gains are no indication of future returns.

As the cost of computing goes down and the amount of data goes up, the likelihood that applications fail due to numerical errors is on the rise. We've seen evidence of this regarding our latest compiler improvements. Numerical differences appeared as part of our automated testing of built-in math functions. Such visibility was unexpected given an overall improvement in the accuracy of floating-point operations. Instead, we found that many of these tests were sensitive to numerical changes even when the underlying computations were more accurate. The lessons we've learned while investigating and addressing these numerical differences are presented in this paper.

We start by characterizing numerical error and proving its existence. From there, we look at how

numerical error relates to methods used to test accuracy. Finally, we see how the standard programming tools and techniques for investigating numerical behavior are impacted by the new compiler improvements.

Understanding Numerical Error

Characterizing the variation in accumulation of numerical error during the course of execution is complicated. Because applications written in G rely on more than just the compiled LabVIEW source code, it is impossible for each version of LabVIEW to produce the same numerical results as a prior version. The same is true of one version of LabVIEW running the same application on different platforms.

Many functions in LabVIEW are built using third-party compilers and incorporate functionality from external numerical libraries at run-time. These tools are platform-specific and offer a spectrum of accuracy in terms of their numeric results. As operating systems and numerical libraries are updated, those modifications are built into newer versions of LabVIEW so that performance expectations on newer systems are met.

Despite these variables, tools exist that characterize numerical error in a practical way. Like noise, error is not avoidable: “You can't stop it. You can only hope to contain it.” Instead, limiting the magnitude of computational error is the goal. Tolerances are chosen such that computations downstream have acceptable resolution.

Machine Epsilon

In IEEE754, multiple floating-point formats are defined. Each format is designed to store a range of real numbers with a predetermined accuracy. In LabVIEW, these formats are referred to as representations. Storing a real number in a numeric with a specific floating-point representation results in ***round-off error*** – the rounding error produced by quantization of the real number in binary.

Because the point 'floats,' the magnitude of the round-off error changes based on the size of the number being stored. Fortunately, a relationship exists between the two. The ratio of the error to its stored value is bounded by a value called ***machine epsilon***. Machine epsilon is a fixed value and is tied to the precision of the floating-point representation. In layman's terms, machine epsilon is the largest error expected when storing values close to 1. More formal definitions exist but this is sufficient for our needs.

Machine epsilon is an excellent basis for estimating numerical error. It easily scales to model the round-off error for values of varying magnitudes. Normalizing data is a process representing an implicit application of machine epsilon. By scaling signal data to magnitudes of at most 1, signal processing algorithms generate comparable numerical results. Even if machine epsilon is not used explicitly in these circumstances, the fundamental concept is being applied.

Machine epsilon has been a constant on the Numerics palette for several releases. In LabVIEW 2010, this constant has been upgraded to support all three floating-point representations – single

(SGL), double (DBL) and extended (EXT). Because machine epsilon depends on the underlying precision, this constant is different for each representation.

Displaying Numerical Data

Machine epsilon is a natural way to measure numerical error. In practice, the epsilon value is much smaller than most data values. This makes detecting round-off error more difficult. Consider the real number 0.1. This value is used as a step size to iterate over intervals such as [1, 10]. It's popular because it is simplifies the math to an extent that we can know the theoretical answer ahead of time.

In knowing the exact solution, we set expectations the computer can't meet. Our method of computing uses symbolic techniques that avoids the round-off error in the floating-point system. We treat 0.1 as 1/10 and manufacture results as if this value is stored exactly. In reality, this is not true. When the decimal string "0 .1" is entered into a numeric control, the string is converted to a value stored in a floating-point numeric.

Example 1: Displaying a Number Stored in Floating-Point

Here, the control shows 0.1 using 16 decimal places while hiding trailing zeros. The value appears to be stored with no numerical error.	
Changing the control to display 17 decimal digits reveals that 0.1 is not the value stored in the computer after all. In fact, this newly displayed value may not be exact either.	

> Lesson #1: Visualizing data stored in floating-point can be misleading. The value you see is modified to meet the properties of the numeric control or indicator. By default, numeric data in LabVIEW is displayed using 6 significant digits.

If 0.1 is not the value stored in the computer, then what is the actual value? The answer is not trivial but, thankfully, we don't have to know it. Here is where machine epsilon comes in handy. Compliance to IEEE754 specifies that 0.1 is represented by a value in the range $(.1-\delta_1, .1+\delta_2)$ where δ_1, $\delta_2 \leqslant .1\times\epsilon$ and ϵ is machine epsilon. Tests checking accuracy downstream focus on this range rather than on the actual error.

The quantization error shown in ***Example 1*** is not just an artifact of the underlying floating-point system. We carry some responsibility because we use decimal (base 10) representation to input and inspect data. Floating-point systems compliant to IEEE754 use binary (base 2) representations. Only numbers that are products of powers of 2 – of the form $\prod 2^{n_i}$ – can be converted between these representations without error. Others, such as $.1 = 2^{-1}\cdot 5^{-1}$, are rounded.

> Lesson #2: Inspecting the accuracy of a value stored in floating-point is in itself inaccurate if it is displayed using a decimal representation.

Before you begin reconfiguring all of the controls and indicators on your front panels to display 20 or more digits, understand that these limitations are mitigated if the methods used to check numerical accuracy incorporate valid tolerances.

Comparing Floating-Point Values

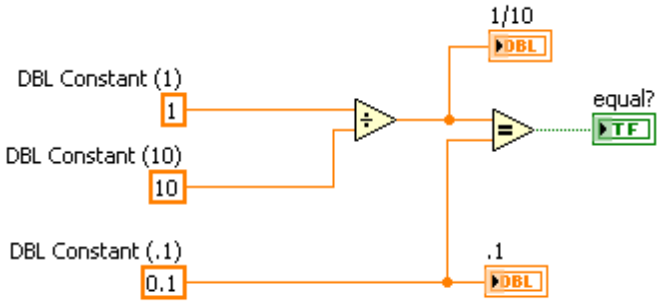
Checking the numerical behavior of your algorithms should employ floating-point comparison. If it doesn't, chances are it won't weather modifications to the application or to the system on which is executes. If the underlying mathematics do not demand a particular tolerance, then some form of machine epsilon should play a role in the comparison process.

Assuming the behavior of floating-point operations to be the same as their integer counterparts leads to a very common but flawed method of comparing floating-point values.

Example 2: Comparing Floating-Point Values Using Equality

The implementation at the right computes 0.1 using division and compares the result to 0.1 stored in a numeric constant (DBL).

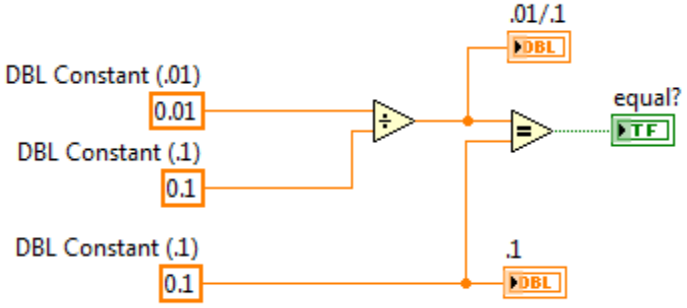
From ***Example 1*** we know that the constant is not as 1/10 but as an approximation. When this diagram is run, **equal?** is TRUE.



If we modify the division operation such that the outcome is still theoretically 0.1, we find that **equal?** is now FALSE.

Was the difference due to the error in storing the division arguments or in the division operation itself or both?

The answer is convoluted and difficult in practice.



Operations that are equivalent in theory do not always preserve that property when executed on computers nor are they guaranteed to produce results identical to the theoretical solution.

> Lesson #3: With very few exceptions, using equality to compare two floating-point values is not appropriate. Such a comparison, if successful on a specific platform or in a given version of LabVIEW, is likely to fail when computing conditions change.

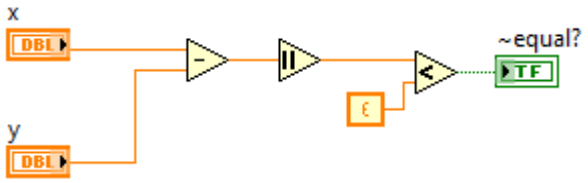
Comparison methods based on machine epsilon navigate the deficiencies found in strict equality. The simplest form of comparison tests for ***absolute error***. Absolute error measures quantities

using unmodified input data and a fixed tolerance.

Example 3: Floating-Point Comparison Using Machine Epsilon

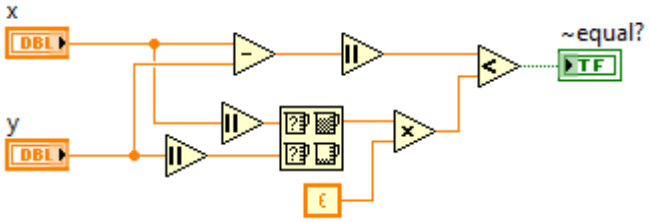
VERSION 1 (GOOD)

The diagram to the right represents a form of absolute error which uses machine epsilon. It is more forgiving than strict equality in ***Example 2*** but is most effective when the largest magnitude of **x**, **y** is close to 1.



VERSION 2 (BETTER)

By scaling machine epsilon based on the inputs, this version adjusts error expectations based on the size of the values. As a result, this implementation is appropriate for a wider range of input values.



You may be wondering why a BEST version is omitted. In general, other types of tests for numerical accuracy are superior such as those computing ***relative error***. While VERSION 2 in ***Example 3*** adjusts the magnitude of the error tolerance, an implementation based on relative error scales parameters other than the tolerance.

Here are two examples of formulas for computing the relative error:

Scalar	$err = \frac{\lvert x-y \rvert}{\lvert x \rvert}$, where x is the *expected value* and y is the *computed value*
Matrix	$err = \frac{\lVert A-A' \rVert_2}{\lVert A \rVert_2}$, where A and A' are $n \times n$ matrices

Relative error scales parameters based on different types of numerical measures depending on the data type and mathematical domain. For such formulas to be effective, they need to test the key (mathematical) characteristics necessary for the algorithm to succeed.

> Lesson #4: Comparing floating-point values involves properly scaled differences and tolerances.

In signal processing, testing a Fast Fourier Transform (FFT) introduces challenges similar to the floating-point comparison of scalars. The output generated by an FFT has two fundamental attributes: a size corresponding to the input size and an array of bin values corresponding to amplitudes of each frequency.

When testing the output of an FFT against an expected result, comparing the size and bin values can be as unforgiving or misleading as using equality to compare floating-point values. In most scenarios, the frequency information as a whole is what's crucial to future computations. Is it

possible that two different FFT results impart the same frequency information for the same input signal? Absolutely! A test comparing attributes claims that the FFT results

$$\{0,1,0,0\} \text{ and } \{0,0,1,0,0,0,0,0\}$$

are not equivalent yet they are equivalent in terms of their spectral content. If the goal is to compare based on spectral characteristics, then alternative formulas are needed. For example,

$$err = \|x - FFT^{-1}(FFT(x))\|$$

computes a ***backward error*** – an estimation of error based on recovering the original input – for the FFT. A complementary form of error is called ***forward error*** – an estimate of error realized in future computations. The latter form is one used in more tests for numerical error. However, once a function becomes significantly complicated, computing the forward error is difficult if not impossible. Backward and forward error represent what is being tested while absolute and relative error refer to how the test is computed.

Lesson #5: What you test for is as important as how you test.

Basic methods for testing numerical accuracy of mathematical functions are available online or in print. Exploiting these references saves time and avoids hours of debugging.

Analyzing Floating-Point Behavior

When proper numerical checks are in place, several benefits are realized:

1. Examining the numerical tests visually is more effective. The numerical error computed as part of an appropriately scaled comparison produces a value that, when displayed, has leading digits relevant to the numerical accuracy.

Unscaled		**Scaled**
0.10000000000000001	versus	2.2204460492503131E-16

2. Detecting significant numerical differences during execution reduces the time spent identifying possible sources.
3. Introducing a new version of LabVIEW or executing your application on a different platform is less likely to alter the automated decisions derived from one or more floating-point computations.

Improving numerical stability of an application won't eliminate errors but it does limit their scope. When an application exploits specific attributes based on key use cases to improve performance, failures occur when these conditions are not satisfied. Finding where these failures take place is crucial to evaluating their significance.

Sometimes changes in one or more components that ship with LabVIEW trigger an existing application to fail even when accuracy is improved. One of the latest compiler optimizations added in LabVIEW 2010 includes more efficient use of processor registers which also improves

numerical accuracy of consecutive floating-point computations. Although rare, algorithms do exist that are so sensitive to numerical precision when implemented that they fail because of the additional accuracy.

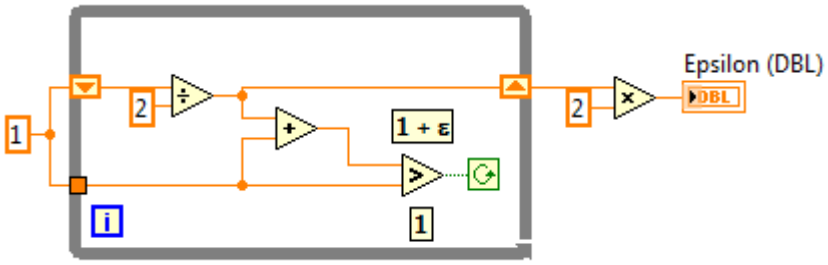
The diagram shown in ***Example 4*** is an implementation of an algorithm for computing machine epsilon. In prior releases of LabVIEW, it correctly computes machine epsilon for the *double* precision (DBL) floating-point representation. It was a viable implementation because the following three conditions were satisfied:

1. Integer numbers such as 1 and 2 are stored exactly in floating-point.
2. Division by 2 introduces no numerical error.
3. Addition by 1 produces a larger value when the sum is fully representable in the numeric precision of the input types.

The fundamental steps of the algorithm are explained on the left.

Example 4: Computing Machine Epsilon

- Initialize with 1.
- Iteratively:
  - Compute 2^{-i}
  - Add 1
  - Stop when the result is not greater than 1
- Compute the prior result



When re-compiled and run in LabVIEW 2010, this implementation produces machine epsilon but for the wrong precision – *extended*. The new compiler optimization altered the numerical behavior.

Investigating a Moving Target

Finding where the compiler changed results sounds like a job for the debugger or execution highlighting. To enable either of these capabilities, the **Allow Debugging** execution property of a VI must be turned on. Doing so changes the execution – the code computes the machine epsilon for *double* precision as it did in prior releases.

> Lesson #6: Compiler optimizations such as register allocation may be disabled when debugging is turned on.

This is not a bug. Debugging (or execution highlighting) requires access to data associated with every wire on the diagram. Data in the processor's register cannot be viewed. It must first be read into memory. This process rounds the value to *double* precision negating the extra precision. Executing this debug version of the code mimics the behavior of previous LabVIEWs where results were not left in registers.

This method failed to reproduce the problem but perhaps a more simplistic approach might. Why

not insert indicators where it made sense to add probes? This may alter behavior too. By adding new wires and terminals, the code gets recompiled such that the allocation of floating-point registers is no longer the same.

> Lesson #7: Seemingly benign changes to a block diagram can reorder floating-point computations behind the scenes and alter numerical results.

Welcome to Heisenberg's "Uncertainty Principle" in action – whatever one observes, one taints. This catch-22 appears insurmountable lest we recall the basis for Lessons #1 and #2. Numerical error is not an exact science. The primary goal is to limit numerical error by understanding its sources.

Knowing When to Say When

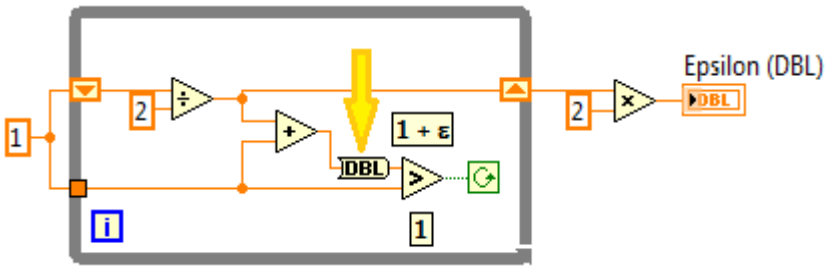
By revisiting the three conditions on which the implementation relied, we find that condition 3 is not satisfied if the compiler keeps the output of the addition operator in a register. In this scenario, the comparison operation that follows performs its computation using inputs that are offspring of extended precision arithmetic.

Examples of algorithms such the one in ***Example 4*** are atypical. However, implementations that do not compare floating-point values correctly exhibit the same sensitivity under subtle numeric changes and are equally difficult to debug.

> Lesson #8: Extra precision is rarely the cause of undesirable numeric results. Chances are a numerical stability issue or an invalid precondition is lurking.

For those cases where preserving the numerical precision of a particular computation is critical, the extended precision gained thru register allocation can be nullified by using a conversion bullet. In LabVIEW2010, the conversion bullets **To Double Precision Float** and **To Single Precision Float** on the Conversion palette round the value on a wire to the corresponding precision even when the value is in a register.

By adding a conversion bullet strategically, the implementation in ***Example 4*** adheres to the original conditions and guarantees the correct machine epsilon is computed.



Applying conversion bullets to solve this problem is safe even when the implementation is saved to a previous version. Prior to LabVIEW 2010, conversion bullets mapping an input to an output of the same type left the value unaltered.

Lesson #9: In rare cases where extra precision is problematic, use the conversion bullets to round the value to its numeric precision on the diagram.

Taking the Next Step

We've presented the foundation for understanding and investigating changes in numerical behavior in LabVIEW 2010. New features in the G compiler are generating code the performs better on multi-core processors and enhances floating-point accuracy in certain circumstances. As a result, existing applications run in the latest version will produce different numerical results than in prior versions.

The knowledge and methods promoted in this paper are designed to raise awareness of common numerical issues and to help you resolve those that impact your application. Numerical differences are a concern even when they are a result of improved accuracy. The key lies in employing proper evaluation techniques so that you can apply a fix that withstands future changes.

If you take time to improve the handling of numerics in your existing applications, you'll reap the rewards when future improvements introduce further numerical differences. Change is inevitable. The destination is multi-core. We've just placed our foot on the accelerator. It's your job to steer.
