the WINNOWER

COMPUTER SCIENCES

# Why bitwise reproducibility matters

KONRAD HINSEN



CORRESPONDENCE:
thewinnower@khinsen.fastmail.net

DATE RECEIVED:
June 10, 2015

KEYWORDS:
reproducible research, software testing, floating-point arithmetic



While reading the final report of the reproducibility workshop at XSEDE14, I noticed a statement that I encounter frequently in discussions about reproducible research:

"One general consensus was that bitwise reproducibility is often an unrealistic expectation"

In the interest of clarity, let me start by pointing out that within the systematic terminology that I am trying to adopt (see this post for an explanation), I will write "bitwise replicability" from now on, as the problem falls into the technical domain (getting the same result from running the same program on the same data) rather than into the scientific one (verifying a result with similar but not identical methods and tools).

The particularity of bitwise replicability is that is almost always brushed aside as "unrealistic", which prevents any discussion about its possible importance in computational science. The main point of this post is to explain why I consider bitwise replicability important, but first of all I need to get the label "unrealistic" out of the way.

"Unrealistic" means more or less "possible in principle but impossible given various real-life contraints", and therefore the term should always be qualified by listing the constraints that make something impossible. In the context of bitwise replicability, which always refers to floating-point computations, the main constraint is that floating-point arithmetic is incompletely specified in most of today's programming languages, and that whatever specification there is is incompletely implemented in many of today's compilers. This is a valid reason for proclaiming bitwise replicability unrealistic for a short-term research project, but it is not an insurmountable barrier on a longer time scale. All we need are tighter specifications and implementations that respect them. That's a lot of work, but not a technical challenge. We know how to do it, but we are not (yet) willing to invest the effort to make it happen.

The main reason why I consider bitwise replicability important is software testing. No matter what precise approach is used for testing, it always involves comparing results of computations, either to a known good result, or to the result of another, presumably more reliable, computation. For any application of computing other than number crunching, comparing results means testing for equality, at the bit level. The results are equal or they aren't. If they aren't, there's a reason. You have to figure out what that reason is, and fix the problem.

If you accept the idea that floating-point operations are only approximate, the notion of a computation having one and only one result disappears, and testing becomes impossible. If two computations lead to similar but slightly different results, how do you decide if this is due to a bug or to some "inevitable" fuzziness of floating-point arithmetic? The answer is that you can't. If you accept that bitwise replicability is not possible, you also accept that rigorous software testing is not possible. For some illustrations of this problem, and some interesting discussion around them, see this post on the

Software Carpentry blog.

The most common counterargument is that numerical methods are only approximate, that floating-point arithmetic is approximate as well, and that the main source of error comes from these two sources. That may or may not be true in any specific situation, as it really depends on what you are computing. But my point is that this statement can only be true if you assume that the implementation of your method contains no mistakes. The amount of error introduced by a bug in the code is completely unbounded. And even if it's small for some particular test run, it can be very large elsewhere. There is not much point in worrying about the error in an approximate numerical method unless you have some confidence in your code actually implementing this method correctly.

In fact, the common counterargument discussed above conflates several sources of error, which can and should be discussed and analyzed separately. A typical numerical computation is the result of several steps, starting from a mathematical model that takes the form of algebraic or differential equations:

1. Construct a computable approximation to the original equations, using techniques such as discretization of continuous quantities.
2. Replace real-numbers by floating-point numbers.
3. Implement the floating-point version in software.

The errors introduced in the first step are the subject of numerical analysis, a well-established domain of applied mathematics. They are well understood for most commonly employed numerical methods. The errors introduced in the second step are rarely discussed explicitly, outside of a small circle of researchers interested in the peculiarities of floating-point arithmetic. The third step should not introduce any errors, and that should be verified by testing. But uncoupling steps 2 and 3 is possible only if our software tools guarantee bitwise replicability.

So why don't today's tools permit this? The reason is a mixture of widespread ignorance about floating-point arithmetic and the desire to get maximum performance. Both come into play in step 2, which is approximating discrete equations for real numbers by discrete equations for floating-point numbers. Most scientific programmers are unaware that this is an approximation that they should understand and control. They just type their real-number equation into a program and expect the computer to handle it somehow. Compiler writers and language specification authors take advantage of this ignorance and declare this step their business, profiting from the many optimization possibilities it offers.

The optimization opportunities come from the fact that a typical real-number equation has a large number of a priori equally plausible floating-point number approximations. Many of the identities for real numbers do not apply to floating-point numbers, for example associativity of addition and multiplication. Where the real-number equation says $a + b + c$, there are three floating-point approximations: (a+b)+c, a+(b+c), and (a+c)+b. For more complex equations, the number of variants quickly becomes important. The results of these variants are not the same, but which one to choose? The choice *should* be made after a careful analysis of the relative precision and performance of each variant. There *should* be tool support to help with this. But what happens in practice, most of the time, is that the choice is made by the compiler, which goes exclusively for performance. Since every compiler optimizes differently, the same program source code yields different results on different platforms. And that's why we don't have bitwise replicability.

To prevent any misunderstanding: I am *not* saying that production-level compiled code needs to ensure bitwise reproducibility across machines. It's OK to have compiler optimization options that introduce platform-specific approximations. But it should be possible to reproduce one unique result identically on all platforms. This result is then the reference against which additional "lossy" optimizations can be tested.