Numerical Replication of Computer Simulations: Some Pitfalls and How To Avoid Them [*]

Theodore C. Belding
Center for the Study of Complex Systems
University of Michigan
Ann Arbor, MI 48109-1120 USA
Ted.Belding@umich.edu
http://www-personal.umich.edu/%7Estreak/

Abstract

A computer simulation, such as a genetic algorithm, that uses IEEE standard floating-point arithmetic may not produce exactly the same results in two different runs, even if it is rerun on the same computer with the same input and random number seeds. Researchers should not simply assume that the results from one run replicate those from another but should verify this by actually comparing the data. However, researchers who are aware of this pitfall can reliably replicate simulations, in practice. This paper discusses the problem and suggests solutions.

1 INTRODUCTION

When we perform computer simulations, such as genetic algorithms, it is often useful to be able to replicate a run exactly, so that those results of the second run that we care about are exactly the same as those of the first. This kind of replication is called *numerical replication* [1]. For instance, if we notice a strange result in a run, it is useful to be able to redo the run exactly, using the same parameter settings and random number seeds, but this time collecting additional data or perturbing the course of the run in order to test hypotheses about what is causing the strange results. Numerical replication can also be used to verify that experimental results are not due to a bug or human error, and to perform regression testing after making changes to a program.

However, a program that uses IEEE standard floating-point arithmetic [13, 6] may produce different results on two different computers, even if the same input and random number seeds are used. In fact, there is no guarantee that it will produce the same results when run twice on the same computer, or even that a subexpression will have the same value when evaluated at two different points during a single run. This is because the calculations may be performed at different precisions each time, and the programmer has little control over what precision is used [21]. This can cause numerical replication to fail unexpectedly. In the worst case, this can lead us to believe that two different sets of results are the same, and thereby cause us to draw incorrect conclusions.

Luckily, if we are aware of these pitfalls, we can reliably avoid them in practice. We should not simply assume that two sets of data are the same because we used the same input and random number seeds; instead, we should always verify this empirically. Furthermore, we should always record the computer platform and run-time and compile-time parameters that we use along with the simulation data. This will make numerical replication easier to achieve. We need tools to make both of these tasks easy and automatic. Finally, we need to compile a knowledgebase of heuristics for achieving numerical replication. In the remainder of this paper, I discuss the problem further and justify these recommendations.

2 THE PROBLEM

Computers perform arithmetic mainly on two kinds of numbers: integers (such as 42) and real numbers (such as 3.14159). There are various possible ways to represent real numbers in a computer; almost all modern computers use binary *floating-point* representations [20, 11, 10, 18]. This representation is essentially the same as scientific notation, except in binary; besides representing real numbers, it can also be used to represent very large integers. Floating-point numbers are represented in the form $(-1)^s \times 1.f \times 2^x$. The part s is the *sign bit* (0 or 1), $1.f$ is called the *significand* (an older term is *mantissa*), f is called the *fraction*, where $0 \leq f < 1$, and x is called the *exponent*. Al-

[*]January 25, 2000 draft; submitted to GECCO 2000. Comments welcome!

```
#include <stdio.h>

int main() {
    double q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

Figure 1: Example program that may produce different results on different computer platforms, from Priest [21]

```
#include <fpu_control.h>

void __attribute__ ((constructor))
enter_fpu_double_mode () {
    (void) __setfpucw ((_FPU_DEFAULT &
        ~_FPU_EXTENDED) | _FPU_DOUBLE);
}
```

Figure 2: Simply compile and link this C code with a program using the `gcc` compiler on x86 or m68k platforms to put the FPU in double-precision mode. Adapted from the g77 manual [22]. Other compilers should provide a similar way to do this. On x86, more needs to be done to completely emulate double precision; see the text for details.

most all computer platforms used today use IEEE 754 standard [13, 6] floating-point arithmetic. The only important exceptions are the Cray X-MP, Y-MP, C90, and J90, the IBM /370 and 3090, and the DEC VAX; most of these are disappearing rapidly [14]. This standard has caused floating-point arithmetic to be much more reliable, predictable, and portable.

However, the standard does not guarantee that a program will produce the same results when run on two different computers [21]. This is because different computers may perform floating point calculations differently, even if the computers all follow the IEEE standard. The standard specifies three different precisions for floating-point arithmetic: *single precision* (32 bits long), *double precision* (64 bits) and *double extended precision* (also called simply *extended precision*, 80 or more bits). Different computer platforms support these precisions to different extents.

We may get different results, for instance, when we run a simulation once on a Hewlett-Packard (HP) PA-RISC workstation running HPUX and once on an Intel x86 PC running Linux or MS Windows. The HP workstation uses single-precision and double-precision floating point arithmetic, while the x86 uses IEEE 80-bit extended-precision floating-point arithmetic by default. (The Motorola 680x0 (m68k) is another CPU family that uses 80-bit extended precision; it was used in the first Macintosh computers.) Figure 1 shows an example program that may produce a different result on each platform, depending on the compiler and the compile-time settings. An HP workstation will print "Equal", while an x86 computer may print either "Equal" or "Not Equal", depending on the compiler and compile-time options that are used [21].

If the results of a simulation depend on many floating-point calculations, this difference in precision may cause the two runs to produce wildly different results. This is particularly likely in simulations of complex systems, such as a genetic algorithm, where the simulation's precise trajectory is highly sensitive to the initial conditions and to the stream of random numbers. Even if the different runs produce the same qualitative results, the numeric results may differ.

This may occur with any program that uses native IEEE floating-point arithmetic, written in any language, on any computer or operating system. Discrepancies may also occur in integer arithmetic, but only if a program makes unwarranted assumptions about the size or representation of integer variables (for example, assuming that C variables of type `int` are 32 bits long).

Both the x86's and the m68k's floating-point unit (FPU) can be switched into "single-precision" or "double-precision" mode (see Figure 2). This solves this particular problem on the m68k. Unfortunately, even when the x86 is in one of these modes, it will still produce different results than an HP or similar workstation would, since its internal registers will still use more bits of precision for the exponents (15 bits instead of 8 or 11) [12]. To reduce the exponent range to be the same as that in "pure" single or double precision, the result must be stored to memory from the x86 FPU's internal registers, and then reloaded from memory into the FPU. This will cause the computation to be two to four times slower than native floating-point arithmetic. If the `gcc` compiler is being used, this can be accomplished by using the `-ffloat-store` compiler option. However, there may still be a discrepancy on the x86 in the last bit of about 10^{-324} because of double rounding, if the floating-point operation is a multiplication or division. To avoid this, one of the operands must be scaled down before the operation by $2^{x_{\max_{\text{extended}}}-x_{\max_{\text{double}}}}$, where $x_{\max_{\text{extended}}}$ is the maximum possible exponent for extended precision and $x_{\max_{\text{double}}}$ is the maximum possible exponent for double precision, and the result must be scaled back up by the same amount afterwards. This additional scaling adds only marginally to the computation time [12].

By using this technique, replication problems can be made much less likely, at the expense of computation speed. However, it will not guarantee that such problems will not occur. In fact, a program may produce different results when run twice on the same computer, even if the same input and random number seeds are used. This is because the results produced by a program depend not only on the computer's floating-point unit and operating system but also on the compiler, the compile-time options, the compile-time and run-time libraries installed, and the input (here I include the date, the run-time environment, and the random number seeds). For instance, the discrepancy may occur if we run a simulation twice on a x86 computer, where the simulation is compiled the first time to store floating-point results to memory, and the second time to keep the results in the FPU's internal registers. Also, the libraries of mathematical functions such as log and sin may produce different results on different platforms [14] and may also differ from version to version on the same platform. (The IEEE standard only contains specifications for the square root function $\sqrt{x}$.) The IEEE standard also does not completely specify the accuracy conversion between binary and decimal representations. It is even technically possible that the results may depend on what other programs are running on the computer, or on bugs in the program, compiler, or libraries — this is especially true if the program is not carefully designed and implemented. Therefore, each time a simulation is run, it is prudent to act as if it were run on a different computer, even if the computer is in fact always exactly the same.

A related issue is that if a floating-point expression occurs more than once in different locations in a program, it may be evaluated to different precisions each time it is used during a single run [21]. For example, on an x86 computer the compiler may choose whether to keep a result in extended precision in the FPU or store it in double precision to memory based on the optimization level, the number of free floating-point registers, whether the result will be used as the argument to a function, and many other factors. (The forthcoming C99 ANSI/ISO C standard will guarantee that if the expression is stored in a variable, the same precision will be used whenever the variable's value is evaluated.) Besides complicating numerical replication, this may cause problems if the program assumes that the expression always evaluates to the same value during the course of a run. The `fcmp` package [3] implements Knuth's [18] suggestions for safer floating-point comparisons, which can be used to avoid this.

Finally, some CPUs, such as the PowerPC, provide an operation called *fused multiply-add* that can perform the operation $\pm ax \pm b$ in a single instruction. If this instruction is used, a different result may be produced than if it is not used, since there is one less rounding step [21]. Also, in expressions such as $\pm ax \pm by$ it is ambiguous which side is evaluated first (and hence rounded). Therefore, this instruction must not be used in certain algorithms, for instance when multiplying a complex number by its conjugate [15, 14]. Unfortunately, many compilers make it difficult for the programmer to specify whether this instruction should be allowed or inhibited in a program.

Guaranteeing that two runs of a program will produce exactly the same results is extremely difficult and may be impossible in practice. Every component which might affect the results would have to be guaranteed to be the same for both runs; none of these components could ever be changed or upgraded unless the new version could be shown to have no effect on the results. On the one hand, determining the version of every component on a computer and recording all of this information with the simulation data would be extremely expensive in time and storage. On the other hand, it will be extremely difficult to weed out false positive results when testing whether two computers have different components: The fact that one of two otherwise identical computers has a copy of the game Quake installed probably will not affect whether a simulation will produce identical results on the two machines, but it will be difficult to prove this. Finally, the date is always changing, and this might have unforeseen effects on a program's behavior. (Consider the recent Y2k problem, or the bug that depended on the phase of the moon [23].)

3 RECOMMENDATIONS

If guaranteeing that we can numerically replicate a run is not an option, what can we do? I suggest that instead of asking how we can guarantee replication, we should ask two different questions: First, what is the worst-case result that can occur because of this problem, and how can we avoid it? Secondly, how can we make numerical replication easier to achieve and more reliable?

The worst thing that can happen when we try to numerically replicate a run is that we mistakenly believe that the replicated results are exactly the same as the original, when they are in fact different. Our main concern should be to avoid this mistake. Luckily, there is an easy way to avoid it: Simply compare the data sets. If they are empirically identical, we are done. (Of course, if we do not record enough data from each run, it is possible that the runs' actual trajectories may be different, even though the data are the same.)

Therefore, we need a set of easy-to-use tools to compare results from two runs, and we should use these tools even if the runs were done on the same computer, as a sanity check. In some cases, where entire files need to match exactly, a utility such as the Unix `diff` command may suffice. In other cases, I suggest using Rivest's [24] MD5 message digest algorithm. This algorithm produces a short string (called a *hash*) that is easy to store with the data that it is computed from. Instead of comparing entire files, only the hash string from each file needs to be compared. If the data files clearly mark comments and other data that we do not need to replicate, such as the date of the run, then it is easy to write a short Perl program [26, 27] to compute an MD5 hash string from a data file, ignoring such extraneous information. (One common convention for marking comments in text files is to put a pound sign '`#`' at the beginning of each comment line.) If it is necessary to ensure that a dataset has not been tampered with in any way, there are cryptographically secure methods, such as signing the data set with PGP [28, 9] or GnuPG [19], or using another message digest algorithm, such as RIPEMD-160 [8, 4]. (MD5 should *not* be used for this purpose [25]!)

Often, using the same input and random number seed will be all that is necessary to numerically replicate a run. Sometimes, however, this will not suffice. In this case, we can almost always replicate the results by tweaking a few special compile-time or run-time parameters (such as what precision the FPU uses). Experience suggests that numerical replication is usually easy to achieve in practice, even though it may be impossible to guarantee. In some cases, it may be necessary to rerun the simulation on the same computer platform that was used originally. For instance, if a simulation is run on an x86 platform using extended precision, it will be difficult to numerically replicate the results on any platform other than an x86 or m68k using extended precision. In addition to the techniques for comparing results mentioned previously, we need a set of heuristics for numerical replication, such as a list of compile-time and run-time parameters that often need to be tweaked. One such heuristic is the technique for emulating double-precision floating-point on x86 computers described in Section 2.

To make numerical replication easier, the compile-time and run-time parameters that were used should be stored with a simulation's results, along with information such as the program and compiler versions, the date, the name of the machine being used, the platform and operating system, etc. In addition to tools for comparing simulation results, we also need tools that make storing this kind of information easy and automatic. (Perl [26] is an example of a program that stores a great deal of configuration information at compile-time; the information is accessible under Unix by running `perl -V`.) A researcher can then use this information when trying to numerically replicate the run. For example, if a simulation is run on an x86 computer using extended precision, it is important to record this fact.

A future release of Drone [2] will include tools for recording and comparing MD5 hashes of data files and for recording compile-time and run-time parameters of simulation programs. I hope that making these tools available will encourage researchers to use them every time they run a simulation.

4 CONCLUSION

In summary, these are problems that everyone doing computer simulations should be aware of, but they are not insurmountable. In practice, a few simple techniques should be sufficient to avoid problems. First, we should never assume that the results from two simulation runs are identical because they used the same parameters and random number seed, even if they are run on the same computer. We should always verify this, either by comparing the relevant results directly or by comparing the MD5 hash strings of the two datasets. This verification process should be made so convenient that there is no reason not to do it. Secondly, we should compile a knowledgebase of likely parameters that can be tweaked to achieve numerical replication, if simply redoing a run with the same input and random seeds does not suffice. Finally, we should always store the compile-time and run-time parameters that we use. We need tools to make this convenient and automatic, as well.

5 FURTHER READING

For a technical discussion of this problem, see Priest [21] and the Java Grande Forum Numerics Working Group's draft report [12]. For a gentle introduction to floating-point arithmetic in general, see Patterson and Hennessy [20] or Goldberg [11]; for a more technical discussion, see Goldberg [10] or Knuth [18]. The IEEE 754 floating-point standard is published in [13]; for a readable account see Cody, et al. [6]. Cody and Coonen [5] give C algorithms to support features of the standard. Kahan and Darcy [17] and Darcy [7] argue that it is undesirable to enforce exact replicability across all computing platforms, and Kahan [16] gives an example of differences in floating-point arithmetic in different versions of `Matlab` on various platforms. Axtell, et al. [1] discuss the differing degrees to which a simulation can be replicated. See Rivest [24] and Robshaw [25] for information on the MD5 message digest algorithm; for a Perl interface to MD5, see Winton [27]. For information on RIPEMD-160, a more secure replacement for MD5, see Bosselaers [4] or Dobbertin [8].

6 ACKNOWLEDGMENTS

I am grateful to the members of the `egcs` and `gcc` mailing lists for answering questions and providing information and references. I also thank the other members of the University of Michigan Royal Road group (John Holland, Rick Riolo, Bob Lindsay, Leeann Fu, Tom Bersano-Begey, and Chien-Feng Huang) for their comments and encouragement.

References

[1] Robert L. Axtell, Robert M. Axelrod, Joshua M. Epstein, and Michael D. Cohen. Aligning simulation models: A case study and results. *Computational and Mathematical Organization Theory*, 1:123–41, 1996. Earlier version available as Santa Fe Institute Working Paper 95-07-065 at `http://www.santafe.edu/sfi/publications/Working-Papers/95-07-065.ps`.

[2] Theodore C. Belding. Drone homepage. `http://drone.sourceforge.net/`, 2000.

[3] Theodore C. Belding. fcmp homepage. `http://fcmp.sourceforge.net/`, 2000.

[4] Antoon Bosselaers. The RIPEMD-160 page. `http://www.esat.kuleuven.ac.be/%7Ebosselae/ripemd160.html`, 1999.

[5] William J. Cody, Jr. and Jerome T. Coonen. Algorithm 722: Functions to support the IEEE standard for binary floating-point arithmetic. *ACM Transactions on Mathematical Software*, 19(4):443–451, December 1993.

[6] William J. Cody, Jr., Jerome T. Coonen, David M. Gay, K. Hanson, David Hough, W. Kahan, R. Karpinski, John F. Palmer, F. N. Ris, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984.

[7] Joseph D. Darcy. Re: floating point. Posting to comp.compilers Usenet newsgroup. `http://www.iecc.com/comparch/article/98-10-120`, October 19 1998.

[8] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160, a strengthened version of RIPEMD. In D. Gollmann, editor, *Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer-Verlag, Berlin, 1996.

[9] Simson Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly, Sebastopol, CA, USA, 1994.

[10] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 23(1):5–48, 1991. `http://www.validgh.com/goldberg/paper.ps` This online version includes the addendum by Priest [21].

[11] David Goldberg. Appendix A: Computer arithmetic. In John L. Hennessy and David A. Patterson, editors, *Computer Architecture: A Quantitative Approach*, pages A13–A38. Morgan Kaufmann, San Francisco, CA, USA, second edition, 1996.

[12] Java Grande Forum Numerics Working Group. Improving Java for numerical computation. Section II.4: Use of floating-point hardware. Public draft. `http://math.nist.gov/javanumerics/reports/jgfnwg-01.html#floating-point`, October 30 1998.

[13] IEEE. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1985.

[14] William Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics, and chemistry. `http://www.cs.berkeley.edu/%7Ewkahan/ieee754status/baleful.ps`, 1996.

[15] William Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. `http://www.cs.berkeley.edu/%7Ewkahan/ieee754status/ieee754.ps`, 1996.

[16] William Kahan. Matlab's loss is nobody's gain. `http://www.cs.berkeley.edu/%7Ewkahan/MxMulEps.pdf`, 1998.

[17] William Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. Talk presented at ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford University. `http://www.cs.berkeley.edu/%7Ewkahan/JAVAhurt.pdf`, 1998.

[18] Donald E. Knuth. Floating-point arithmetic. In *The Art of Computer Programming*, volume II: *Seminumerical Algorithms*, chapter 4.2, pages 214–264. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[19] W. Koch. GnuPG homepage. `http://www.gnupg.org/`, 1999.

[20] David A. Patterson and John L. Hennessy. Arithmetic for computers: Floating point. In *Computer*

Organization and Design: The Hardware/Software Interface, chapter 4.8, pages 275–301. Morgan Kaufmann, San Francisco, CA, USA, second edition, 1997.

[21] Douglas Priest. Differences among IEEE 754 implementations. Addendum to Goldberg [10]. `http://www.validgh.com/goldberg/addendum.html` It is also included in the online version of Goldberg [10] at `http://www.validgh.com/goldberg/paper.ps` To be published in the next release of Sun's *Numerical Computation Guide*, 1997.

[22] GNU Project. *Using and Porting GNU Fortran*. 1999. `http://gcc.gnu.org/onlinedocs/g77_toc.html` or execute `info g77` on a computer that has a recent version of `g77` installed.

[23] Eric S. Raymond. Entry "phase of the moon". In *The New Hacker's Dictionary*, pages 326–27. MIT Press, Cambridge, MA, USA, second edition, 1993. Current version of the entry is at `http://www.tuxedo.org/%7Eesr/jargon/html/entry/phase-of-the-moon.html`.

[24] Ronald L. Rivest. The MD5 message-digest algorithm. RFC 1321, IETF, 1992. `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1321.txt`.

[25] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. Bulletin 4, RSA Laboratories, November 12 1996. `ftp://ftp.rsa.com/pub/pdfs/bulletn4.pdf`.

[26] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly, Sebastopol, CA, USA, 1996. Perl homepage: `http://www.perl.com/`.

[27] Neil Winton. `MD5.pm`. MD5 Perl module. `http://www.perl.com/CPAN-local/modules/by-module/MD5/`, 1996.

[28] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995. PGP is available for commercial use from Network Associates, Inc. at `http://www.nai.com/` or for non-commercial use in the US from MIT at `http://web.mit.edu/network/pgp.html`.