

The Limits of Reproducibility in Numerical Simulation

Modern computational simulation's increasing and mainly speed-oriented use of HPC systems often conflicts with the goal of making research reproducible. Indeed, the simulations that result from HPC use often behave reproducibly in only a limited way. As a discussion of this phenomenon's technical background describes, the problems entailed will be very difficult to overcome.

In recent years, making research results reproducible has become increasingly important. This applies not only to the two traditional methods used in science—theory and experiment—but also to their newly established counterpart, the computational simulation. Here, the term “reproducibility” is basically understood from a software developer's viewpoint—that is, a newly developed algorithm should be described in a way that lets other programmers implement it on their environment in a semantically exactly equivalent manner. On a classical hardware platform, such as a von Neumann architecture, this implies that the results are also reproducible (unless the code contains true random elements). That is, two successive runs of the algorithm with identical input data will yield identical output data, at least if the executions are performed under identical hardware and operating system environments. Early steps in this direction can be found in the work of Donald Knuth,¹ which was followed and extended by many researchers, including Jonathan Buckheit and David Donoho² and Matthias Schwab and his colleagues,³ and more recently by James Quirk,⁴ Patrick Vandewalle and his colleagues,⁵ and a special

issue of *CiSE*.⁶ In particular, this work offers suggestions for software tools that drastically simplify the job of creating and publishing reproducible results, and even includes a legal framework for this task addressing—among other things—intellectual property issues in this context.

On the other hand, there's also a trend in the numerical simulation science toward increasingly complex models and stricter accuracy requirements that, in turn, force users to apply the most up-to-date high-performance computing (HPC) systems to obtain the desired results in an acceptable time. This includes, in particular, the use of massively parallel hardware and specialized software that's highly tuned to maximize the algorithms' speed and to utilize the hardware's technical possibilities. In extreme cases, such as highly complex optimization problems, using stochastic methods—such as genetic algorithms⁷—might be required to obtain good approximate solutions because deterministic algorithms can't produce good results fast enough. Similarly, for numerical integration of functions on high-dimensional domains, deterministic methods are often rather slow and might force users to revert to Monte Carlo methods,⁸ which also have built-in randomness. Evidently, in such cases users can achieve reproducibility only if the random sequences involved in the computations are replaced by pseudo-random sequences whose seeds aren't changed from one program execution to the next. In practice, this is feasible and not uncommon but

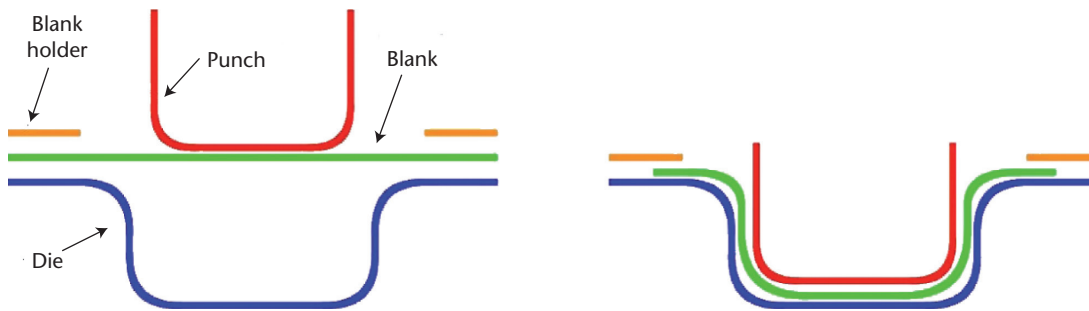


Figure 1. Example application: the deep drawing process for deforming metal sheets. The image shows a cross section of tools (die, punch, and blank holder) and the workpiece (blank) at the beginning (left) and end (right) of a deep drawing process.

in a certain sense it corrupts the underlying basic theory.

In many cases, however, we must deal with problems whose computational complexity isn't so extreme that we have to use stochastic algorithms; most can instead be tackled by classical, deterministic algorithms. Given reproducibility theory as described in the research mentioned earlier, we could then possibly create reproducible results. Of course, it would be necessary to invoke the various programs exactly as in the original computation, but then, according to the theory, "[a]n author whose research involves scientific computations on a UNIX computer can easily create reproducible documents."³

This is where we must start looking at reproducibility from a second point of view—namely, the perspective of the software's end user. Such users aren't typically interested in semantics details or other programming issues, but rather want reproducibility in the sense that two runs of the code with identical input data should produce identical output data. As mentioned above, this isn't usually a problem on sequential systems, but unfortunately it often doesn't work in an HPC environment. In this latter sense, the request for reproducibility isn't fulfilled by many common efficient and proven high-performance algorithms.

In the early days of HPC, close cooperation between software developers and end users was common—and it wasn't uncommon for these two roles to be filled by the same person. A helpful side effect of this close cooperation was, and is, that the two parties usually had a thorough understanding of each other's methods and the difficulties encountered; in particular, the end user understood the reasons for a lack of reproducibility and was able to live with this phenomenon and draw useful conclusions from the computed data.

In recent years, however, HPC systems have become more generally available, and are now

frequently used by nonspecialists—such as engineers with strong backgrounds on the experimental side but little knowledge of numerical mathematics and computational simulation. These people tend to buy standard off-the-shelf HPC software and often have little or no interaction with the software developers. For these end users, it's usually quite difficult to understand and correctly interpret the occurrence of nonreproducible results simply because they don't have the necessary background.

Here, my goal is to elucidate the nonreproducibility phenomenon and explain why it's often considered not to be a problem at all from a mathematical viewpoint and why it's hardly ever discussed. However, I also endeavor to raise awareness among mathematical and software development communities of the fact that their "customers" who want to use these algorithms find this nonreproducibility a problem, and that they don't know how to handle the results.

An Application Example

As an example, let's consider the simulation of a deep drawing process⁹ for the deformation of metal sheets, a standard method used, for example, in the automotive and packaging industry. Figure 1 provides a schematic representation of this process. At the beginning (left side) a plane sheet of metal (the *blank*) is lying on the *die* (the lower part of the tool). A second part of the tool, the *blank holder*, is moved downward from above the blank. This tool exerts a certain force so that the outer part of the workpiece is fixed between the die and blank holder and can't move upward. Then, the deep drawing process itself begins, in which the third part of the tool, the *punch*, is moved downward with a prescribed force. The material of the blank is thus drawn downward into the die and plastically deformed (right side).

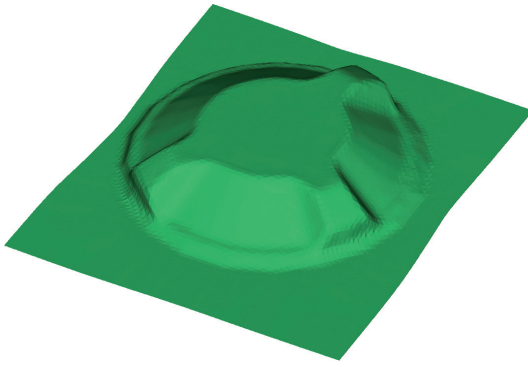


Figure 2. The part to be manufactured. Manufactured from a 0.6-mm thick sheet of steel, the part is about 350-mm long and 400-mm wide. The finite elements used for the simulation have an edge length of 4 mm.

To simulate the deformation of a plane sheet to a real car body part with such a deep drawing process, my colleagues and I at GNS use the Indeed program (www.gns-mbh.com/?id=indeed), a software system specially designed to simulate sheet metal forming based on the finite element method (FEM). As an example, consider the part depicted in Figure 2, which is supposed to be manufactured from a 0.6-millimeter-thick sheet of steel. The part is about 350-mm long and 400-mm wide. The finite elements used for the simulation have an edge length of 4 mm.

At GNS, we repeatedly performed the simulation with different degrees of parallelism. Specifically, we varied the number of processors between one and four, and executed the simulation with four processors twice. All other input data were kept constant. We then observed the program's

behavior and compared the output data. In this case, we chose the output data to be a variable that's particularly important in practice—namely, the local change of the sheet thickness. This variable is a crucial criterion in the process of deciding whether the manufactured part will in practice be suitable for its intended application. The user will typically have two threshold values for the sheet thickness. If the computed result is below the smaller of these values, the part will be considered insufficient and the deformation process must be modified to change this behavior. If the computed values are above the larger of the two values, the workpiece will be accepted, and if the computation produces results between the two given values then additional investigations are required to determine how to proceed.

Even though the algorithms used don't contain any stochastic elements, the results always differed from each other in all respects. In particular, it's not uncommon for results to move from one side of one threshold value to the other as the simulation is repeated. Thus, the way in which the engineers must proceed in developing their tools and the details of the tool control can also vary, depending on how the algorithms are executed on the given hardware.

To illustrate the variation of the program's behavior, let's look at two specific aspects. The first is the way in which its automatic time-step control works. The program decomposes the period of time required for the process in reality into individual time steps, and the simulation is performed in these discrete time steps. The sizes of the first two time steps are determined by the user (we always choose the same values for all our experiments); the following step lengths are automatically chosen in a deterministic way by the program depending on the convergence behavior of the previous step.

Figure 3 shows a visualization of the corresponding data in a graph that illustrates the share of the already-simulated process time with respect to the total process time over the number of time steps. It's evident that the behavior differs from program run to program run. In particular, the two curves corresponding to the two runs with four processors (that is, for two executions of the same program with absolutely identical input data in every respect) significantly differ from each other.

We've also compared the computed changes in sheet thickness. Here, I present only some selected values. Table 1 shows the computed extreme values of the sheet thickness change. Even in this small dataset, the significant differences are

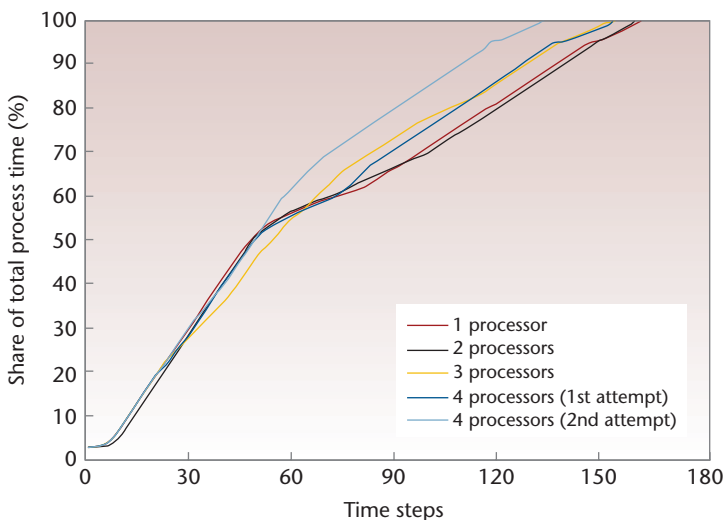


Figure 3. Behavior of the time-step control. The behavior differs across program runs.

Table 1. Computed extrema of the sheet thickness change.

Description of the simulation (no. of processors)	Minimal value of the sheet thickness change (%)	Maximal value of the sheet thickness change (%)
1	−29.21	+9.54
2	−29.34	+9.14
3	−29.04	+9.02
4 (1st attempt)	−29.04	+8.98
4 (2nd attempt)	−29.09	+8.96

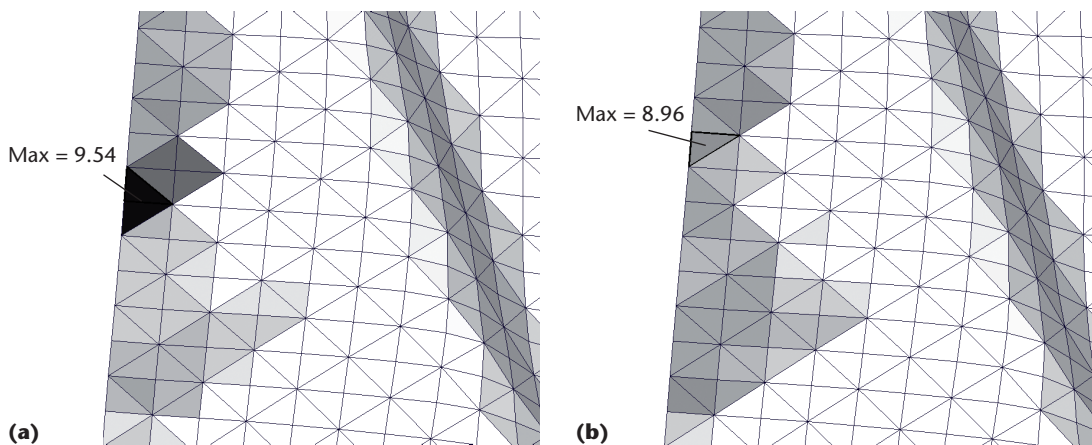


Figure 4. Location of the computed maxima of the sheet thickness change. (a) The simulation with one processor. (b) The second run of the simulation with four processors. The darker the element is colored, the larger the corresponding sheet-thickness change. Elements colored in white have a sheet thickness change of less than 8.5 percent.

clearly visible. In particular, the maximal values vary more than the minimal values. The locations where these extremal values arise on the blank are exemplified in Figure 4 by a comparison of only two program runs. The figure shows the same section of the blank in both cases, and it's evident that not only the extremal value itself changes but also its location.

A detailed investigation of the program source and the intermediate results revealed that the reason for the differences that we observed was the nondeterministic behavior of the Pardiso subroutine¹⁰ that was linked to the program code as a component of the Intel Math Kernel Library (<http://software.intel.com/en-us/intel-mkl>). Pardiso is a parallel direct (that is, noniterative) solver for sparse linear systems that, due to its reliability and efficiency, is popular and frequently used. In particular, it's a natural subroutine for a finite element program such as Indeed, where many such systems must be solved.

In a complex simulation, such a situation is common. The required code is so voluminous and consists of subroutines from so many different areas that most researchers or research teams don't

have enough time and/or specialized know-how to develop and implement the complete code on their own. Thus, users typically have to rely on more or less well-documented library routines. Of course, end users typically don't have any influence on the way in which these routines were written, and so in a case like this, the reproducibility is at least doubtful. Therefore, to be positively sure about reproducibility, a close cooperation with the developers of the employed libraries is essential.

I'll now discuss how such effects can arise and what can be done—primarily by the researchers who write the subroutine libraries—to avoid them.

The Mathematical Background

To illustrate the mathematical background of the nonreproducibility described earlier, it's not necessary to consider an extremely complicated high-performance algorithm. Rather, it's sufficient to look at a completely obvious approach for the solution of a simple problem that, strictly speaking, can be interpreted as an approximation method only because it's executed on a computer in finite precision arithmetic. As will be immediately clear, the phenomenon under investigation

is based on absolutely elementary properties of computer arithmetic. Therefore, it's inherent to almost every numerical software system unless explicit countermeasures are applied.

A Simple Example Problem

Our prototypical example problem is the computation of the scalar product:

$$s = \sum_{i=1}^n x_i y_i \quad (1)$$

of two n -dimensional vectors x and y . This problem can be interesting in its own right, but it's also a part of many advanced procedures of applied mathematics, such as the conjugate gradient method¹¹ for the solution of linear systems. For us, it's a good example because its simple structure doesn't hide the problem's true essence.

Imagine that the dimension n of the vectors is quite large. This is a common situation in scientific computing; values of the order $n > 10^6$ aren't a rarity. Thus, the calculation of the scalar product is associated with a non-negligible computational effort. However, this task is highly suitable for a parallelization. To keep the description as simple as possible and to avoid losing the focus on the relevant aspects by introducing unnecessary technical detail, we assume all processors work equally fast. It then makes sense to divide the n summands of the sum to be computed in Equation 1 into p parts, where p is the number of available processors. Without loss of generality, we let p be a divisor of n . (Otherwise, we could extend the vectors x and y by adding more components, all of which having the value zero.) Then, we can give the l th processor ($l = 1, 2, \dots, p$) the task of computing the partial sum,

$$s_l = \sum_{i=(l-1)n/p+1}^{ln/p} x_i y_i \quad (2)$$

of the required scalar product. Afterward, we'll ask one of the processors to determine the final result:

$$s = \sum_{i=1}^p s_l. \quad (3)$$

Fixed Number of Processors

With this algorithm, the critical point is the way in which the final sum (Equation 3) is assembled. For reasons of efficiency, a typical implementation won't add up the summands here in their natural order; rather we'll use the order in which the individual processors finish the computations of

their respective partial sums. Thus, if, for example, all processors except processor #2 have completed their calculation, the processor associated with the final summation will already add up their intermediate results. As soon as processor #2 has finished, its result s_2 will be added to that result, and the final result will be available. If we chose the natural order of summation, the computation of the final sum (Equation 3) would have to be interrupted until processor #2 had completed its job, and only then could all other s_l results be added up. Thus, the time required overall for the computation of the scalar product s would become larger because significantly more operations would be required after the end of processor #2's computations than in the previous procedure.

If we now run through the same algorithm a second time with identical input data, it's by no means certain that the individual processors complete their respective tasks in the same order as in the program's first execution. This is because each processor typically must perform not only the computational tasks that we've prescribed, but also additional tasks, such as those associated to the operating system. For these additional tasks, our proper computations will be interrupted. It's generally impossible, at least for typical users of the hardware platform in question, to predict both when such an interruption will occur and how long the interruption will last. Thus, we typically won't be able to control the order in which the partial results from the respective processors will arrive—that is, the order in which the partial sums will be assembled to form the final sum. Now, however, standard computer arithmetic uses floating-point numbers with a finite precision. Therefore, the summation is commutative but—because of the potentially different rounding of the intermediate results—not associative. Hence, the final result s of the summation (Equation 3) depends on the order in which the values s_l are processed.

For a concrete example, let's use $n = 8$ and $p = 4$, so that each partial sum s_l consists of exactly two summands. The x_j and y_j are given by $x_1 = 10^{12}$, $x_2 = 0$, $x_3 = 10^{-8}$, $x_4 = 0$, $x_5 = -10^{12}$, $x_6 = 0$, $x_7 = 10^{-8}$, $x_8 = 0$, and $y_j = 1$ ($j = 1, 2, \dots, 8$). Thus the partial sums s_l amount to $s_1 = 10^{12}$, $s_2 = s_4 = 0$, $s_3 = -10^{12}$. If now the s_l arrive in the order s_1 , s_3 , s_2 , s_4 , then we first compute the intermediate result $z_1 := s_1 + s_3 = 10^{12} - 10^{12} = 0$, then $z_2 := z_1 + s_2 = 0 + 10^{-8} = 10^{-8}$, and finally the final result

$$s = z_2 + s_4 = 10^{-8} + 10^{-8} = 2 \cdot 10^{-8}. \quad (4)$$

In the case of the order s_1, s_2, s_3, s_4 , however, the first intermediate result is $z'_1 := s_1 + s_2 = 10^{12} + 10^{-8}$. Standard IEEE double-precision arithmetic has an accuracy of only approximately 16 decimal digits. Thus, the computer can't represent z'_1 exactly and must round it to $z''_1 = 10^{12}$, which will then be used in the subsequent calculations. Hence, we come to the next intermediate result, $z'_2 := z''_1 + s_3 = 10^{12} - 10^{12} = 0$ (without additional rounding errors), and finally to the final result,

$$s = z'_2 + s_4 = 0 + 10^{-8} = 10^{-8}, \quad (5)$$

which differs from the value found in Equation 4 by 50 percent.

We can easily prevent the problem from occurring under the conditions discussed here; to this end, it suffices to begin the execution of the summation (Equation 3) only when all summands s_j have really been computed. Then it's easily possible to sum up the s_j in their natural order or in any other deterministic way (which might be preferable, considering the propagation of the rounding errors). This procedure asserts that the same operations will always be executed in the same order, no matter how much time each processor takes for the completion of its subtask. It follows that the results produced by the program's first run can be safely reproduced in any later run of the program with the same input data. However, this reproducibility has the price of additional waiting times that are introduced even though they're unnecessary from the purely computational viewpoint. The only purpose of these waiting times is an artificial synchronization of the subprocesses, and it's evident that we must expect an increase of the runtime required by the program. In our very simple example here, the loss of speed will be small, but in more complex situations, the effects might be much more severe.

Varied Number of Processors

It's not uncommon for a software developer to be asked by a program's end users to explain certain phenomena that can be seen in the results. In such a situation, it's often necessary for the software developer to execute the program with the user's input data once again to be able to see the intermediate results. For various reasons, it might be necessary to re-execute the program on a different number of processors than originally used. In this case, additional complications arise.

To illustrate this problem, let's again look at the computation of the scalar product. If we modify the number of processors in the algorithm

indicated above, we change the distribution of the individual summands across the partial sums s_j . An increase in the number of processors leads to a decrease of the number of summands in each partial sum and vice versa. Thus, the way in which the summands are arranged to obtain the final result also changes due to the implicit introduction of parentheses, so the propagation of the rounding errors will also happen in a different manner.

This phenomenon can also be illustrated using the concrete example from the previous subsection. If we choose $p = 2$ instead of $p = 4$, then we must assign the task of summing up four numbers to each of the two processors. This gives the partial sums $s_1 = 10^{12} + 10^{-8}$, which once again can't be exactly represented and must be replaced by the rounded value $s_1^* = 10^{12}$ and $s_2 = -10^{12} + 10^{-8}$, which also can be used only in the rounded form $s_2^* = -10^{12}$. Thus, we obtain the final result $s^* = s_1^* + s_2^* = 10^{12} - 10^{12} = 0$, which coincides with neither of the values derived in Equations 4 and 5.

A way out of the problem here isn't as evident as above. A potential approach that has, for example, been used in a similar form in the latest version of the Pardiso library,¹² is based on introducing another parameter $p^* \geq p$ in addition to the number p of processors actually in use. This new parameter p^* is supposed to be interpreted as an upper bound for the admissible values of p . The algorithm will then be implemented as if p^* processors were available—that is, the overall task will be distributed across p^* virtual processors according to the scheme indicated above. Each of the p physically existent processors will then first have to do the work of one virtual processor. Having completed this subtask, the processor then proceeds, if required, with the execution of a subtask of another virtual processor that hasn't been dealt with yet. In this way, we can ensure the reproducibility of the partial results. To obtain the final result in a reproducible way, we must then assemble the partial results appropriately as described earlier.

As a concrete example, we could choose $p^* = 4$. Setting $p = 2$, each physical processor must then perform the job of two virtual processors. So, for example, the first processor would compute the partial sums $s_1 = x_1y_1 + x_2y_2$ and $s_3 = x_5y_5 + x_6y_6$, whereas the second processor would form the partial sums $s_2 = x_3y_3 + x_4y_4$ and $s_4 = x_7y_7 + x_8y_8$. As in our previous case, the final summation $s = s_1 + s_2 + s_3 + s_4$ must then be done in a reproducible manner, too.

Although this procedure can assert the reproducibility of results when the number of processors changes, it does have some disadvantages. First, there's an additional loss of efficiency because it's not always possible to distribute the virtual processors' work to the physically existent processors in a balanced way. This leads to certain physical processors being idle while others are still busy doing their computations. Thus, the available hardware resources aren't used in an optimal manner. A second disadvantage is that the parameter p^* must be chosen. If you discover later that more than p^* processors should be used, you again lose the reproducibility. Hence, in practice, this approach can be implemented only with restrictions.

A more feasible alternative for our simulation problem seems to be the recently published algorithm by Yong-Kang Zhu and Wayne Hayes.¹³ Their method sums arbitrarily long streams of floating-point numbers in such a way that the final result will be the correctly rounded value of the input data's exact sum, no matter what order the input data arrive in (except for the case of an overflow at some intermediate stage). Hence, in the standard situation (no intermediate overflow) it will always produce the same output, giving us the desired reproducibility.

This algorithm requires an amount of additional runtime and memory that would typically be acceptable. Other reproducible methods exist,^{13,14} but they tend to be substantially slower and/or require high amounts of memory, which make them difficult to accept by users looking for high-performance algorithms. Of course, all of these algorithms solve the problem only if the nonreproducibility is due *only* to differently propagated rounding errors in summations; they don't help if it's caused by other operations. In such cases, we can revert to other techniques such as interval arithmetic¹⁵ or uncertainty quantification^{16,17} to understand how the problem reacts to the propagation of rounding errors and similar influences.

Keep in mind, however, that interval arithmetic usually comes with an undesired increase in runtime and that uncertainty quantification is usually performed by probabilistic methods—and thus will only give results that must be interpreted in a stochastic sense. Another option is to use techniques such as multiple-precision arithmetic, fixed-point arithmetic, double-double-precision summation, or self-compensated summation. All of these ideas have been discussed elsewhere,¹⁸ but the authors of that paper explicitly state that, “The

‘exact’ ... reproducibility, i.e., identical numerical results ... on different number of processors ... is impossible in our view, and is not our goal.”

The mathematical background of non-reproducibility is simple and well known in the scientific computing community. Because it's so ubiquitous in numerical high-performance algorithms, we've learned to live with it and to draw the right conclusions from the variations in the output data. Thus, because the problem is so well understood in this peer group, a common assumption is that it's more of a feature than a problem; it's thus possible to conclude that reproducibility isn't a requirement that absolutely must be enforced.


After all, an algorithm's end users typically have no insight into how the code has been executed and therefore can't judge whether, for example, rounding errors have been propagated unfavorably. Hence, if an algorithm is executed more than once with unchanged input data and produces different sets of output data, then each of these results is equally plausible and the only possible interpretation of the output is that of an approximation of the correct value with a certain error. Thus, there's no reason to believe that any of the program's results are indeed the exact solution (or to believe that they're a particularly bad approximation). In this sense, nonreproducible results actually have a certain “instructional” effect because they prevent the user from thinking, “If the computer *always* gives me the same result then this *must* be the true exact solution.” This point is sometimes interpreted as a positive feature of nonreproducibility, thus reinforcing the belief that this phenomenon isn't a problem.

Nevertheless, a nonreproducible behavior can be seriously disturbing for end users, such as when a user's task explicitly requires reproducibility for some reason, the program's author wants others to be able to check the code's integrity, or the target algorithm's output data are used as input data for a subsequent part of the program. In the latter case, you might specifically think of situations where the later part of the program is unstable, so small changes in input data can lead to significant differences in the output or even make the difference between convergence and divergence of an iterative method. Often, such an instability can be traced back to the fact that the simulated system itself is unstable in reality. In such cases, you can once again argue that the simulation's seemingly unpredictable

behavior only reflects the real system's corresponding property and that the user should be cautious when drawing conclusions from the computation's results. Still, it's frequently difficult to get the message across to users that a program doesn't contain an error in the classical sense of the word, especially if the end users are inexperienced or not well educated in numerical mathematics and scientific computing.

In any case, it's the mathematicians' and software engineers' job to convey an awareness for this problem to the end users of such methods who, after all, often have their roots in other disciplines and might not have much relevant, specialized knowledge in mathematics or computer science. In particular, they should make it clear that the lack of reproducibility is typically a price that must be paid for speeding up the algorithm. It would of course be desirable to discuss this fact in the basic mathematical modules for non-mathematical students, but in practice, this will probably be possible only in a limited number of specialized lectures for only a fraction of the target audience. Thus, the software developers must also

- mention this topic in their documentations, and
- write routines that can reproduce results exactly for those users who need such a feature.

Finally, this aspect should also be attended to in the workshops and training courses offered by many HPC centers, on whose hardware the corresponding programs are typically executed. 

References

1. D.E. Knuth, "Literate Programming," *Computer J.*, vol. 27, no. 2, 1984, pp. 97–111.
2. J.B. Buckheit and D.L. Donoho, "WaveLab and Reproducible Research," *Wavelets and Statistics*, A. Antoniadis and G. Oppenheim, eds., Springer Verlag, 1995, pp. 55–81.
3. M. Schwab, M. Kärrenbach, and J. Claerbout, "Making Scientific Computations Reproducible," *Computing in Science & Eng.*, vol. 2, no. 6, 2000, pp. 61–67.
4. J.J. Quirk, "Computational Science: 'Same Old Silence, Same Old Mistakes'; Something More is Needed," *Adaptive Mesh Refinement—Theory and Applications*, T. Plewa, T. Linde, and V.G. Weirs, eds., Springer Verlag, 2005, pp. 3–28.
5. P. Vandewalle, J. Kovačević, and M. Vetterli, "What, Why and How of Reproducible Research in Signal Processing," *IEEE Signal Processing*, vol. 26, no. 3, 2009, pp. 37–47.
6. S. Fomel and J.C. Claerbout, eds., "Reproducible Research," special issue, *Computing in Science & Eng.*, vol. 11, no. 1, 2009, pp. 5–40.
7. C.W. Ahn, *Advances in Evolutionary Algorithms*, Springer Verlag, 2006.
8. M. Evans and T. Swartz, *Approximating Integrals via Monte Carlo and Deterministic Methods*, Oxford Univ. Press, 2000.
9. H. Tschaetsch, *Metal Forming Practise*, Springer Verlag, 2006.
10. O. Schenk and K. Gärtner, "Solving Unsymmetric Sparse Systems of Linear Equations with Pardiso," *J. Future Generation Computer Systems*, vol. 20, 2004, pp. 475–487.
11. G. Meurant, *The Lanczos and Conjugate Gradient Algorithms*, SIAM, 2006.
12. O. Schenk, M. Bollhöfer, and R.A. Römer, "On Large-Scale Diagonalization Techniques for the Anderson Model of Localization," *SIAM Rev.*, vol. 50, no. 1, 2008, pp. 91–112.
13. Y.-K. Zhu and W.B. Hayes, "Algorithm 908: Online Exact Summation of Floating-Point Streams," *ACM Trans. Mathematical Software*, vol. 37, no. 3, 2010; doi:10.1145/1824801.1824815.
14. N.J. Higham, "The Accuracy of Floating Point Summation," *SIAM J. Scientific Computing*, vol. 14, 1993, pp. 783–799.
15. R.E. Moore, R.B. Kearfott, and M.J. Cloud, *Introduction to Interval Analysis*, SIAM, 2009.
16. L. Biegler et al., eds., *Large-Scale Inverse Problems and Quantification of Uncertainty*, John Wiley & Sons, 2011.
17. O.P. Le Maître and O.M. Knio, *Spectral Methods for Uncertainty Quantification*, Springer Verlag, 2010.
18. Y. He and C.H.Q. Ding, "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications," *J. Supercomputing*, vol. 18, no. 3, 2001, pp. 259–277.

Kai Diethelm is a developer for mathematical software at GNS (Gesellschaft für Numerische Simulation) mbH in Braunschweig, Germany, and an adjunct professor at the Institut Computational Mathematics, Technische Universität Braunschweig. His research interests include applied mathematics, in particular in numerical methods for the solution of differential equations. Diethelm has a PhD in computer science and a habilitation degree in mathematics from Universität Hildesheim. Contact him at diethelm@gns-mbh.com.



Selected articles and columns from IEEE Computer Society publications are also available for free at <http://ComputingNow.computer.org>.



TIMELY, ENVIRONMENTALLY FRIENDLY DELIVERY

DIGITAL EDITIONS

Keep up on the latest tech innovations with new digital editions from the IEEE Computer Society. At **more than 65% off regular print prices**, there has never been a better time to try one. Our industry experts will keep you informed with the latest technical developments in a format that's timely and environmentally friendly.

- Easy to Save. Easy to Search.
- Email notification. Receive an alert as soon as each digital edition is available.
- Two formats. Choose the enhanced PDF edition OR the web browser-based edition.
- Quick access. Download the full issue in a flash.
- Convenience. Read your digital edition anytime—at home, work, or on your mobile.
- Digital archives. Subscribers can access the digital issues archive dating back to January 2007.

From software architecture to security and networking, these magazines help you stay ahead of the competition:

- *Computer*—our flagship magazine with broad technical coverage
- *IT Professional*—critical IT topics
- *IEEE Software*—practitioner-oriented trends and applications
- *IEEE Security & Privacy*—computer security and data privacy
- *IEEE Internet Computing*—emerging and evolving Internet technologies

Interested? Go to www.computer.org/digitaleditions to subscribe and see sample articles.





stay connected.

Keep up with the latest IEEE Computer Society publications and activities wherever you are.

Follow us on Twitter, Facebook, and Linked In.



@Computer Society, @ComputingNow



facebook

facebook.com/IEEEComputerSociety
facebook.com/ComputingNow



IEEE Computer Society, Computing Now

IEEE  computer society