PERSPECTIVE

doi:10.1038/nature10836

The case for open computer programs

Darrel C. Ince[1], Leslie Hatton[2] & John Graham-Cumming[3]

Scientific communication relies on evidence that cannot be entirely included in publications, but the rise of computational science has added a new layer of inaccessibility. Although it is now accepted that data should be made available on request, the current regulations regarding the availability of software are inconsistent. We argue that, with some exceptions, anything less than the release of source programs is intolerable for results that depend on computation. The vagaries of hardware, software and natural language will always ensure that exact reproducibility remains uncertain, but withholding code increases the chances that efforts to reproduce results will fail.

The rise of computational science has led to unprecedented opportunities for scientific advance. Ever more powerful computers enable theories to be investigated that were thought almost intractable a decade ago, robust hardware technologies allow data collection in the most inhospitable environments, more data are collected, and an increasingly rich set of software tools are now available with which to analyse computer-generated data.

However, there is the difficulty of reproducibility, by which we mean the reproduction of a scientific paper's central finding, rather than exact replication of each specific numerical result down to several decimal places. We examine the problem of reproducibility (for an early attempt at solving it, see ref. 1) in the context of openly available computer programs, or code. Our view is that we have reached the point that, with some exceptions, anything less than release of actual source code is an indefensible approach for any scientific results that depend on computation, because not releasing such code raises needless, and needlessly confusing, roadblocks to reproducibility.

At present, debate rages on the need to release computer programs associated with scientific experiments[2–4], with policies still ranging from mandatory total release to the release only of natural language descriptions, that is, written descriptions of computer program algorithms. Some journals have already changed their policies on computer program openness; *Science*, for example, now includes code in the list of items that should be supplied by an author[5]. Other journals promoting code availability include *Geoscientific Model Development*, which is devoted, at least in part, to model description and code publication, and *Biostatistics*, which has appointed an editor to assess the reproducibility of the software and data associated with an article[6].

In contrast, less stringent policies are exemplified by statements such as[7] "*Nature* does not require authors to make code available, but we do expect a description detailed enough to allow others to write their own code to do similar analysis." Although *Nature*'s broader policy states that "...authors are required to make materials, data and associated protocols promptly available to readers...", and editors and referees are fully empowered to demand and evaluate any specific code, we believe that its stated policy on code availability actively hinders reproducibility.

Much of the debate about code transparency involves the philosophy of science, error validation and research ethics[8,9], but our contention is more practical: that the cause of reproducibility is best furthered by focusing on the dissection and understanding of code, a sentiment already appreciated by the growing open-source movement[10]. Dissection and understanding of open code would improve the chances of both direct and indirect reproducibility. Direct reproducibility refers to the recompilation and rerunning of the code on, say, a different combination of hardware and systems software, to detect the sort of numerical computation[11,12] and interpretation[13] problems found in programming languages, which we discuss later. Without code, direct reproducibility is impossible. Indirect reproducibility refers to independent efforts to validate something other than the entire code package, for example a subset of equations or a particular code module. Here, before time-consuming reprogramming of an entire model, researchers may simply want to check that incorrect coding of previously published equations has not invalidated a paper's result, to extract and check detailed assumptions, or to run their own code against the original to check for statistical validity and explain any discrepancies.

Any debate over the difficulties of reproducibility (which, as we will show, are non-trivial) must of course be tempered by recognizing the undeniable benefits afforded by the explosion of internet facilities and the rapid increase in raw computational speed and data-handling capability that has occurred as a result of major advances in computer technology[14]. Such advances have presented science with a great opportunity to address problems that would have been intractable in even the recent past. It is our view, however, that the debate over code release should be resolved as soon as possible to benefit fully from our novel technical capabilities. On their own, finer computational grids, longer and more complex computations and larger data sets—although highly attractive to scientific researchers—do not resolve underlying computational uncertainties of proven intransigence and may even exacerbate them.

Although our arguments are focused on the implications of *Nature's* code statement, it is symptomatic of a wider problem: the scientific community places more faith in computation than is justified. As we outline below and in two case studies (Boxes 1 and 2), ambiguity in its many forms and numerical errors render natural language descriptions insufficient and, in many cases, unintentionally misleading.

The failure of code descriptions

The curse of ambiguity

Ambiguity in program descriptions leads to the possibility, if not the certainty, that a given natural language description can be converted into computer code in various ways, each of which may lead to different numerical outcomes. Innumerable potential issues exist, but might include mistaken order of operations, reference to different model versions, or unclear calculations of uncertainties. The problem of ambiguity has haunted software development from its earliest days.

Ambiguity can occur at the lexical, syntactic or semantic level[15] and is not necessarily the result of incompetence or bad practice. It is a natural consequence of using natural language[16] and is unavoidable. The

[1]Department of Computing Open University, Walton Hall, Milton Keynes MK7 6AA, UK. [2]School of Computing and Information Systems, Kingston University, Kingston KT1 2EE, UK. [3]83 Victoria Street, London SW1H 0HW, UK.

BOX 1

The United Kingdom Meteorological Office produces (in conjunction with the University of East Anglia's Climatic Research Unit) the downloadable and widely used gridded temperature anomaly data sets known as HadCRUT and CRUTEM3. Yet even such a high-profile data set, developed by an organization with a good standard of software development[34], contained errors that would have been more quickly identified and rectified had the underlying code been readily available.

In 2009, on examining the available data sets and the description of the algorithm[35], J.G.-C. identified a number of errors (the software he used to check the meteorological database is available upon request). One set of errors was procedural, and involved incorrect computation of historical average temperatures in a number of records in New Zealand and Australia. The Meteorological Office confirmed the errors, showed that they had resulted in errors up to 0.2 °C (either warmer or cooler) in the average temperature for Australia and New Zealand in some years before 1900, and issued an update to CRUTEM3. Two other errors occurred in the coding of the calculation of station errors (an estimate of the error in any average temperature reading). When corrected, a minor reduction in station errors resulted, improving the accuracy of the data. So, although these implementation problems did not lead to serious errors in the temperature data sets, they highlight the difficulty of translating a natural-language description (even with some formulae expressed mathematically) into code.

These errors do not in any way reflect badly on the original authors. The code rewriting simply plays the part of peer review and it is normal to find such errors. Indeed, the discovery of such errors in 'working' software is exceedingly common in all computing, even when the software has been in use for a considerable time. This was emphatically demonstrated in a seminal IBM study[36], demonstrating that fully a third of all the software failures in the study took longer than 5,000 execution years (execution time indicates the total time taken executing a program) to fail for the first time.

problem is regarded as so axiomatic that its avoidance or minimization is routinely taught at the undergraduate level in computing degrees. Nor is the study of ambiguity confined to the classroom. Active research continues on the use of tools for the detection of ambiguity[17], the avoidance of ambiguity in major projects[18], and the clarification of the intended functions of computer programs[15].

One proposed solution to the problem of ambiguity is to devote a large amount of attention to the description of a computer program, perhaps expressing it mathematically or in natural language augmented by mathematics. But this expectation would require researchers to acquire skills that are only peripheral to their work (set theory, predicate calculus and proof methods). Perhaps worse, investment of effort or resources alone cannot guarantee the absence of defect[19]. A recent study[20] of a tightly specified, short, simply expressed algorithm whose semi-mathematical specification was supplemented by example outputs showed that major problems still arose with large numbers of programs individually implemented to this specification. In short, natural language descriptions cannot hope to avoid ambiguous program implementations, with unpredictable effects on results.

Errors exist within 'perfect' descriptions

Let us assume for a moment that a researcher, perhaps trained—as are computer scientists—to think of computer algorithms as mathematical objects, and fully versed in the formal semantics of software description, has managed to describe a computer program perfectly in some notation. Unfortunately, even such a description would not ensure direct or indirect reproducibility, because other forms of error or ambiguity (unrelated to natural language) are likely to creep in, leading to potentially serious uncertainties (Box 2).

BOX 2

As discussed, unambiguous descriptions are no guarantee of reproducibility. One example from the geological literature makes the point[37]. This study compared nine different commercial implementations of the same seismic data-processing algorithms, developed independently. Several sources of ambiguity were successfully excluded, the same data set was used, the signal-processing algorithms used were unambiguously specified in mathematics, and the same programming language was used (Fortran 77). The individual companies followed industry standards in code implementation.

Approximately 200,000 lines of code were exercised in each of the packages in a 14-stage pipeline for which the output of each stage was the input to the next. The signal-processing algorithms used would be familiar to many scientists—such as Wiener deconvolution, acoustic wave equation solutions, fast Fourier transforms and numerous common statistical procedures.

The initial stage involved reading 32-bit pressure data from tapes recorded in a marine environment. During the processing pipeline, the agreement between the results of each package declined from the six significant figures present in the input data to only between one and two in the final output. These data, however, were used by geologists to site extremely expensive marine drilling rigs and could "fundamentally affect the conclusions reached as to the nature of potential hydrocarbon accumulations" [37]. Furthermore "it seems reasonable to infer that the primary source of disagreement is indeed software error"[37]. Even porting other seismic software between different architectures using the same input data lost two out of six significant places[12]. On the positive side, correction of the programming errors found during developer feedback led to considerably improved agreement.

Although conducted some years ago, the study is just as relevant today. Fortran 77 is still in use in one dialect or another in scientific research, the same software assurance procedures are still widely used, and scientific programmers are still people, subject to human fallibility.

First, there are programming errors. Over the years, researchers have quantified the occurrence rate of such defects to be approximately one to ten errors per thousand lines of source code[21].

Second, there are errors associated with the numerical properties of scientific software. The execution of a program that manipulates the floating point numbers used by scientists is dependent on many factors outside the consideration of a program as a mathematical object[22]. Rounding errors can occur when numerous computations are repeatedly executed, as in weather forecasting[23]. Although there is considerable research in this area, for example in arithmetic and floating point calculations[24–27], algorithms[28], verification[29] and fundamental practice[30], much of it is published in outlets not routinely accessed by scientists in generic journals, such as *Computers & Mathematics with Applications*, *Mathematics in Computer Science* and the *SIAM Journal on Scientific Computing*.

Third, there are well-known ambiguities in some of the internationally standardized versions of commonly used programming languages in scientific computation[13]. Monniaux[22] describes an alarming example relating to implementation of software features:

"More subtly, on some platforms, the exact same expression, with the same values in the same variables, and the same compiler, can be evaluated to different results, depending on seemingly irrelevant statements (printing debugging information or other constructs that do not openly change the values of variables)."

This is known as an order-of-evaluation problem and many programming languages are subject to its wilful ways. Ironically, such execution

ambiguity is quite deliberate and is present to allow a programming language compiler more flexibility in its optimization strategy. And even when programs are simple, or developed by the largest software companies, such errors remain surprisingly common: numerical ambiguity led Microsoft to declare in 2010 and reaffirm in September 2011, that the treatment of floating point numbers in its popular Excel spreadsheet "...may affect the results of some numbers or formulas due to rounding and/or data truncation." (http://support.microsoft.com/kb/78113).

Perfection is no guarantee of reproducibility

Finally, even if a computer program could be unambiguously described and implemented without error, other problems can arise in machine deployment whereby the results from identical code often diverge when hardware and software configurations are changed[22]. So even perfection in one's own software environment does not guarantee reproducibility. As a result, to maximize the chances of reproducibility and consistency, not only would we urge code release, but also a description of the hardware and software environment in which the program was executed and developed.

Challenges are no excuse for closed code

Nature's policy on code release implies that algorithmic descriptions using mathematical specifications, equations, formal algorithmic descriptions or pseudocode (simplified version of complete code) may be required. But there is no guarantee that such tools can avoid ambiguity[20], and even if they could, we have shown above that implementation and numerical errors—possibly compounded by differences in machine architecture—will still arise. So, even if complete code is made available, exact replication or even reproduction of central results may fail. A reasonable observer might therefore ask why code should be made available at all. Our response is that the alternative is far worse. Keeping code closed ensures that potential uncertainties or errors in a paper's conclusions cannot be traced to ambiguity, numerical implementation, or machine architecture issues and prevents testing of indirect reproducibility. Although it is true that independent efforts to reproduce computational results without recourse to the original source code constitute an important approach, the all-too-common treatment of code as a black box unnecessarily slows and impedes valid efforts to evaluate model results. We therefore regard the non-availability of code as a serious impediment to reproducibility.

Potential barriers and proposed solutions

There are a number of barriers to the release of code. These include a shortage of tools that package up code and data in research articles; a shortage of central scientific repositories or indexes for program code; an understandable lack of perception of the computational problems with scientific code leading to the faulty assumption that program descriptions are adequate (something we address in this article); and finally that the development of program code is a subsidiary activity in the scientific effort.

A modest proposal

An effective step forward would be for journals to adopt a standard for declaring the degree of source code accessibility associated with a scientific paper. A number of simple categories illustrate the idea:

- Full source code: full release of all source code used to produce the published results along with self-tests to build confidence in the quality of the delivered code, as is the case with Perl modules in the CPAN archive, for example (http://cpan.org).
- Partial source code: full release of source code written by the researcher accompanied by associated documentation of ancillary packages used, for example commercial scientific subroutine libraries.
- Marginal source code: release of executable code and an application programming interface to allow other researchers to write test cases.
- No source code: no code at all provided.

This hierarchy of disclosure would alert both the readers and authors of a journal article to the fact that the issue is important and would highlight the degree to which results might be reproduced independently. There remain, however, some potential stumbling blocks, a number of which can easily be resolved using existing facilities.

Intellectual property rights

Clearly, if there is evidence of commercial potential or use, such as a patent or some copyright, then there is a problem. It is difficult to see how a journal might deal with this without substantial financial commitment to independent testing under a non-disclosure agreement or possibly even the purchase of commercial rights. Perhaps the simplest solution is for a journal to flag the software as 'No source code' (ideally giving the reasons) until such time as the source code can be included, either because the code goes into the public domain or is released under some free licence. Such a designation simply says that, for the moment, the results are not reproducible with the authors' own source code, and that testing of the main results must proceed with independent approaches.

Limited access

Researchers may not have access to at least some of the software packages that are used for development. We suggest that this would not be a problem for most researchers: their institutions would normally provide such software. If it were to be a problem, then a journal could mark a publication as 'Partial source code'. The release of the code, even without the software environment required for compilation and execution, would still be valuable in that it would address issues such as dissection and indirect reproducibility (see above) and would enable rewriting using other programming languages.

Procedure

Adopting the simple disclosure of the availability of source code will help make it clear to the readership of a journal that this is an important issue, while also giving them an idea of the degree of code release. However, we would further suggest that journals adopt a standard that specifies that supplementary material supporting a research article must describe each of the released modular components of any software used. *Nature* editors and referees are already empowered to include an appraisal of code in their judgement about the publication potential of the article, and this practice should be more widely advertised and supported. A good example of this approach is the way that the journal *Geoscientific Model Development* asks authors to describe their program code.

Logistics

Over the past two decades, the open-source community has solved the logistics of releasing and storing code while maintaining a cooperative development environment. SourceForge (http://www.sourceforge.net/) is an excellent example. Founded in 1999, it is a web-based source-code repository which acts as a free centralized location for developers working on open-source projects. It currently hosts around 300,000 projects and has over two million registered users. Not only does it store source code but also it provides access to version control information, project wikis (websites that are easily modifiable by its users) and database access. We urge funding agencies to investigate and adopt similar solutions.

Packaging

There are a number of tools that enable code, data and the text of the article that depends on them to be packaged up. Two examples here are *Sweave* associated with the programming language R and the text-processing systems LaTeX and LyX, and *GenePattern-Word RRS*, a system specific to genomic research[31]. *Sweave* allows text documents, figures, experimental data and computer programs to be combined in such a way that, for example, a change in a data file will result in the regeneration of all the

research outputs. *GenePattern-Word RRS* is similar in that it enables an author to link text, tables and figures to the analysis and data that yielded the results, reported in a word-processed document; it also allows further experimentation (for example, additional analyses can be carried out). It is still early days, however, and localized solutions are emerging at the grassroots level. Donoho and co-workers, for example, have developed software packages that allow anyone with access to the *Matlab* programming language and development environment to reproduce figures from their harmonic analysis articles, inspect source code, change parameters and access data sets[32].

Steps to implementation

Our thesis is that journal and funding body strictures relating to code implementations of scientific ideas are now largely obsolete. We have suggested one modest path to code availability in this article. There are a number of further steps that journals, academies and educational organizations might consider taking:

- Research funding bodies should commission research and development on tools that enable code to be integrated with other elements of scientific research such as data, graphical displays and the text of an article.
- Research funding bodies should provide metadata repositories that describe both programs and data produced by researchers. The Australian National Data Service (http://www.ands.org.au/) which acts as an index to data held by Australian research organizations, is one example of this approach.
- Journals should expect researchers to provide some modular description of the components of the software that support a research result; referees should take advantage of their right to appraise software as part of their reviewing task. An example of a modular description can be seen in a recent article published in *Geoscientific Model Development*[33].
- Science departments should expand their educational activities into reproducibility. Clearly such teaching should be relevant to the science at hand; however, courses on statistics, programming and experimental method could be easily expanded and combined to include the concept of reproducibility.

Received 9 May 2011; accepted 5 January 2012.

1. Schwab, M., Karrenbach, M. & Claerbout, J. Making scientific computations reproducible. *Comput. Sci. Eng.* **2,** 61–67 (2000).
2. Barnes, N. Publish your computer code: it is good enough. *Nature* **467,** 753 (2010).
3. McCafferty, D. Should code be released? *Commun. ACM* **53,** 16–17 (2010).
4. Merali, Z. ...Error. *Nature* **467,** 775–777 (2010).
5. Hanson, B., Sugden, A. & Alberts, B. Making data maximally available. *Science* **331,** 649 (2011).
6. Peng, R. D. Reproducible research and biostatistics. *Biostatistics* **10,** 405–408 (2009).
   This work provides a succinct and convincing argument for reproducibility—*Biostatistics* is at the forefront of ensuring that code and data are provided for other researchers.
7. Devil in the details. *Nature* **470,** 305–306 (2011).
8. Pedersen, T. Empiricism is not a matter of faith. *Comput. Linguist.* **34,** 465–470 (2008).
9. Donoho, D. L. An invitation to reproducible computational research. *Biostatistics* **11,** 385–388 (2010).
10. Raymond, E. S. *The Cathedral and the Bazaar* (O'Reilly, 2001).
11. He, Y. & Ding, C. H. Q. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *J. Supercomput.* **18,** 259–277 (2001).
12. Hatton, L. *et al.* The seismic kernel system—a large scale exercise in Fortran 77 portability. *Softw. Pract. Exper.* **18,** 301–329 (1988).
13. Hatton, L. The T experiments: errors in scientific software. *IEEE Comput. Sci. Eng.* **4,** 27–38 (1997).
14. Hey, T., Tansley, S. & Tolle, K. (eds) *The Fourth Paradigm: Data-Intensive Scientific Discovery* (Microsoft Press, 2009).
15. Gervasi, V. & Zowghi, D. On the role of ambiguity in RE. In *Requirements Engineering: Foundation for Software Quality* (eds Wieringa, R. & Persson, A) 248–254 (Springer, 2010).
16. van Deemter, K. *Not Exactly: In Praise of Vagueness* (Oxford University Press, 2010).
17. Yang, H., Willis, A., De Roeck, A. & Nuseibeh, B. Automatic detection of nocuous coordination ambiguities in natural language requirements. In *Proc. IEEE/ACM Intl Conf. on Automated Software Engineering* 53–62 (ACM, 2010).
18. de Bruijn, F. & Dekkers, H. L. Ambiguity in natural language software requirements: a case study. In *Requirements Engineering: Foundation for Software Quality* (eds Wieringa, R & Persson, A.) 233–247 (Springer, 2010).
19. Pfleeger, S. L. & Hatton, L. Investigating the influence of formal methods. *IEEE Computer* **30,** 33–43 (1997).
20. van der Meulen, M. J. P. & Revilla, M. A. The effectiveness of software diversity in a large population of programs. *IEEE Trans. Softw. Eng.* **34,** 753–764 (2008).
21. Boehm, B., Rombach, H. D. & Zelkowitz, M. V. (eds) *Foundations of Empirical Software Engineering: the Legacy of Victor R. Basili* (Springer, 2005).
22. Monniaux, D. The pitfalls of verifying floating-point computations. *ACM Trans. Programming Languages Systems* **30** (3), 1–41 (2008).
23. Thomas, S. J., Hacker, J. P., Desagne, M. & Stull, R. B. An ensemble analysis of forecast errors related to floating point performance. *Weather Forecast.* **17,** 898–906 (2002).
24. Revol, N. Standardized interval arithmetic and interval arithmetic used in libraries. *Proc. 3rd Intl Congr. Mathematical Software* 337–341 (2010).
25. Rump, S. M., Ogita, T. & Oishi, S. Accurate floating point summation. Part 1: Faithful rounding. *SIAM J. Sci. Comput.* **31,** 189–224 (2009).
26. Rump, S. M. Accurate and reliable computing in floating-point arithmetic. *Proc. 3rd Intl Congr. Mathematical Software* 105–108 (2010).
27. Badin, M., Bic, L., Dillencourt, M., &. Nicolau, A. Improving accuracy for matrix multiplications on GPUs. *Sci. Prog.* **19,** 3–11 (2011).
28. Pan, V. Y., Murphy, B., Qian, G. & Rosholt, R. E. A new error-free floating-point summation algorithm. *Comput. Math. Appl.* **57,** 560–564 (2009).
29. Boldo, S. & Muller, J.-M. Exact and approximated error of the FMA. *IEEE Trans. Comput.* **60,** 157–164 (2011).
30. Kahan, W. Desperately needed remedies for the undebuggability of large floating-point computations in science and engineering. *IFIP/SIAM/NIST Working Conf. Uncertainty Quantification in Scientific Computing* (Springer, in the press).
31. Mesirov, J. P. Accessible reproducible research. *Science* **327,** 415–416 (2010).
32. Donoho, D. L., Maleki, A., Rahman, I. U., Shahram, M. & Stodden, V. Reproducible research in computational harmonic analysis. *Comput. Sci. Eng.* **11,** 8–18 (2009).
    Donoho and fellow researchers have been at the forefront of reproducibility for many years; this article reviews their work, including facilities for code presentation.
33. Yool, A., Popova, E. E. & Anderson, T. R. Medusa-1.0: a new intermediate complexity plankton ecosystem model for the global domain. *Geosci. Model Develop.* **4,** 381–417 (2011).
    An example of an article from a journal that asks for code release for model description papers and encourages code release for other categories of paper.
34. Easterbrook, S. M. & Johns, T. C. Engineering the software for understanding climate change. *Comput. Sci. Eng.* **11,** 65–74 (2009).
35. Brohan, P., Kennedy, J. J., Harris, I., Tett, S. F. B. & Jones, P. D. Uncertainty estimates in regional and global observed temperature changes: a new dataset from 1850. *J. Geophys. Res.* **111,** D12106 (2006).
36. Adams, E. N. Optimizing preventive service of software products. *IBM J. Res. Develop.* **28,** 2–14 (1984).
37. Hatton, L. & Roberts, A. How accurate is scientific software? *IEEE Trans. Softw. Eng.* **20,** 785–797 (1994).

Acknowledgements We thank D. Hales of the Department of Design at the Open University and P. Piwak of the Department of Computing for pointing out some reproducibility work outside computing. J.G.-C. is grateful to I. Goz for pointing out the calculation errors in the CRUTEM data from the Meteorological Office.

Author Contributions D.C.I., L.H. and J.G.-C. contributed to all aspects of this article.

Author Information Reprints and permissions information is available at www.nature.com/reprints. The authors declare no competing financial interests. Readers are welcome to comment on the online version of this article at www.nature.com/nature. Correspondence and requests for materials should be addressed to D.C.I. (d.c.ince@open.ac.uk).