The Case for Open Computer Programs

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

Increasingly more and more complex products are being generated by researchers, for example software, very large data sets and movies. Such products provide publication problems for research journals. We concentrate on software and argue that anything less than release of the actual source code used is inadequate for any scientific results that depend on computation. To this end, we review some of the copious literature of ambiguity and error in software development and describe some relevant case studies in climate and seismological software. We outline some simple steps to effect necessary change.

Introduction

This is an exciting time for scientific research: larger and larger computers are enabling theories to be investigated which were thought almost impossible a decade ago, robust hardware technologies are enabling data to be collected in the most inhospitable environments, larger quantities of data are being collected and an increasingly rich set of software tools have become available to analyse computer-generated data. However, there is a problem: that of reproducibility. This article attempts to examine this problem in the context of openly available program code. Our view is that anything less than release of the actual source code used is inadequate for any scientific results that depend on computation

There has been a recent debate concerning the release of program source code associated with scientific experiments[1,2,3] , with views still ranging from total release to the release only of algorithmic descriptions.

A recent example of a journal asking for code release is that of a change in policy at *Science* which has now added code to the list of items that should be supplied by an author[4]. Other journals moving towards code availability include *Geoscientific Model Development*, which asks authors to provide the paper itself and a user manual with a strong recommendation for code release, and *Biostatistics*, which has appointed an editor to assess the reproducibility of the software and data associated with an article[5].

[1] Dept of Computing Open University, Walton Hall, Milton Keynes, UK. MK7 6AA. (d.c.ince@open.ac.uk)
[2] School of Computing and Information Systems, Kingston University, Kingston on Thames, UK. KT1 2EE. (lesh@oakcomp.co.uk)
[3] 83 Victoria St., London SW1H 0HW. (jgc@jgc.org)

In contrast, support for releasing only descriptions of relevant software in the body of an article or the methods section is exemplified by statements such as[6]:

> *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.

This article represents a response to this policy which we believe does not address problems that arise within e-science. Although our arguments are focussed on the implications of this statement it is worth pointing out that it is symptomatic of a wider problem: the scientific community places more faith in the results of computation than is justified by current understanding of the central role ambiguity in its many forms and numerical errors play in such computations.

Much of the debate about data and code transparency has focused on issues that are associated with the philosophy of science, error validation and research ethics[7,8]. We do not wish to rehearse them, but make some new points concerning the need for code release that draw on computing research.

Of primary relevance to our discussion here are dissection and understanding. Code, even if not run by another researcher, provides an invaluable resource for these purposes. This will resonate with any practising scientist faced with trying to understand a poorly specified program and any contributor to the open source movement[9]. Code also provides building blocks for other researchers who want to extend the research work that it was used to support.

Of equal importance is the need to address reproducibility. We note that two types of reproducibility are relevant: direct reproducibility and indirect reproducibility. The former involves the recompilation and rerunning of the code on, say, a different combination of hardware and systems software, in order to detect numerical computation problems[10,11] and interpretation problems found in programming languages[12].

An example of indirect reproducibility occurs where a researcher may want to independently validate something other than the code, for example a set of equations. Here the researcher may want to check that implementation aspects of the original work do not invalidate the original result before starting time-consuming reprogramming; to extract and check the detailed assumptions made by the researcher; and to run their own code against the original to check for statistical validity and explain any discrepancies.

Any debate over reproducibility must of course be tempered by the undeniable benefits afforded by the explosion of Internet facilities, raw speed and data-handling capability that has occurred as a result of major advances in computer technology[13]. These have presented science with a great opportunity to address problems which would have been inconceivable even in the recent past. It is our view, however, the debate over code release is *essential* to resolve as soon as possible in order to garner the full benefits. On their own, finer computational grids, longer and more complex computations and larger datasets while highly attractive to scientific researchers, do not cast light on underlying computational uncertainties of proven intransigence and may even accentuate them.

A Case-study of Difficult Reproducibility

One of us (JGC) detected two errors in software in 2009 that had been developed by the United Kingdom's Meteorological Office. This organization has a high standard of software development[14].

The Meteorological Office produces (in conjunction with the University of East Anglia's Climatic Research Unit) downloadable gridded temperature anomaly datasets known as HadCRUT and CRUTEM3.

On examining the available datasets and the description of the algorithm[15], a number of errors were identified. One set of errors were procedural, but two indicate defects in the software implementation of one of the algorithms in the paper.

These latter errors highlight the difficulty in translating an English-language description (albeit with some formulae expressed mathematically) into software. The first error involved incorrect computation of ‘normal values’ (historical average temperatures) in a number of records in Oceania. The Meteorological Office confirmed the errors and issued an update to CRUTEM3.

The other two errors relate to the calculation of station errors (an estimate of the error in any average temperature reading). The station error was incorrectly calculated because of two different programming errors.

The forensic effort in detecting these errors was large. The only way to verify the implementation was to write appropriate software from scratch and then try to understand the differences which arose.

This does not in any way reflect badly on the original authors. The 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 some very considerable time. This was emphatically demonstrated in a seminal IBM study[16], demonstrating that fully a third of all the software failures in the study took longer than 5,000 execution *years* to fail for the first time.

The Failure of Code Descriptions

The curse of ambiguity

Ambiguity in program descriptions has haunted software development from its earliest days. Such ambiguity can occur at the lexical, syntactic or semantic level[17]; this problem is regarded as so axiomatic that its avoidance or minimization is routinely taught at the undergraduate level in computing degrees.

It is still a major research area in computing in which, for example, the use of tools for the detection of ambiguity are constructed[18] the avoidance of ambiguity in major projects studied[19] and analyses that aid the requirements specification process established[17]. Ambiguity in some written work is not a result of incompetence or bad practice, but is a natural consequence of using natural language[20]; it is unavoidable.

It should also be noted that computer scientists regard a computer program as a mathematical object: a whole branch of computing is devoted to the formal semantics of software. In spite of this, the execution of a program that manipulates the type of floating point numbers used by scientists is dependent on a whole host of factors outside the consideration of a program as a mathematical object[21].

One proposed solution to the problem in ambiguity is to pay a large amount of attention to the description of a computer program; perhaps expressing it mathematically or in natural language augmented by mathematics. However, one objection to scientists employing mathematical specifications is that it would require researchers to acquire skills that are only peripheral to their work (set theory, predicate calculus and proof methods). Perhaps worse, such a solution offers no guarantees of the absence of defect[22]. A recent study[23] that examined software diversity issues discovered, as a by-product, problems with the interpretation of a very small, semi-mathematical specification of a simple computer program.

Even a mathematical specification or a specification consisting of mathematics and natural language converted into software produces results which are highly susceptible to *numerical* error; this is still the subject of considerable research[24,25,26,27].

Errors exist even with ‘perfect’ descriptions

Errors continue to intrude even if the description is ‘perfect’. First, there are programming errors. Over the years, researchers have quantified the occurrence rate of such defects in the approximate range 1-10 per thousand lines of source code[28].

Second, there are also errors associated with the numerical properties of scientific software, where, for example, rounding errors can occur when a large number of computations are repeatedly executed, for example in weather forecasting[29]. While there is considerable research in this area, for example in the area of algorithms[30], in the area of verification[31] and in fundamental practice[32], much of it is published in outlets not routinely accessed by scientists in generic journals such as *Computers and Mathematics with Applications*, *Mathematics in Computer Science* and *SIAM Journal on Scientific Computing*.

Third, there are well-known ambiguities in some of the internationally standardised versions of commonly used programming languages in scientific computation[12]. An alarming example of a particularly subtle form problem is described by Monniaux[21]:

> "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. This executional ambiguity is quite deliberate and is present to allow a programming language compiler more flexibility in its optimisation strategy.

Even if you interpret the Nature policy on code release as covering algorithmic descriptions using pseudo code, mathematical specifications employing equations or algorithmic descriptions there are still the problems detailed in this section associated with numerical accuracy; moreover, there is no guarantee that tools such as pseudo code and other algorithmic descriptions do not give rise to ambiguities[23]. Because of this we would regard the non-availability of code as a serious impediment to reproducibility.

An exemplifying case study

One particular case study makes the above points forcefully. The study compared 9 different commercial implementations of the same seismic data processing algorithms, developed independently[33]. In this particular study, several of the above sources of ambiguity were successfully excluded; the same dataset was used; the signal processing algorithms used were unambiguously specified in mathematics; and the same programming language was used (Fortran 77). The individual companies used software processes which would be considered of a "high standard" in the sense described earlier in this article.

The results were alarming to say the least. Approximately 200,000 lines of code were exercised in each of the packages in a 14 stage pipeline where the output of each stage was the input to the next. The signal processing algorithms used would be familiar to many scientists – 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. Starting with the six significant figures of single precision floating point arithmetic in the input data, four significant figures of agreement were excised from the final output leading to variations between the package results in the second and sometimes first significant figure. However these data are used by geologists to site extremely expensive marine drilling rigs and the particular geological features of interest (unconformity gas traps) required around 3 significant figures to delineate accurately. Perhaps even worse, the variations between the nine different outputs were shown by a process of feedback to be unequivocally related to programming errors *which had remained undiscovered in the respective packages for several years* and the differences between the outputs were non-random. Even porting exactly the same software between different architectures using the same input data lost 2 out of 6 significant places[11].

Although conducted some years ago, the study is just as relevant today. The language 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 scientific programmers and subject to human fallibility.

Similar latent defects also surfaced in an air-traffic control system, again of the order of 200,000 lines of code and developed using mathematical specifications[22].

Even when programs are very much simpler such errors remain surprisingly common. As recently as 2010 (updated Sept 2011) Microsoft issued a warning that because of the implementation of floating point numbers in the popular Excel spread-sheet this ‘may affect the results of some numbers or formulas due to rounding and/or data truncation’ (`http://support.microsoft.com/kb/78113`, accessed Oct 17 2011). When document and design ambiguities are thrown into the mix, the problem of validation to the level we associate with conventional experiments in the natural sciences, is challenging to say the least.

Perfection is no guarantee of reproducibility

Finally, the problems reported here are not just confined to software issues, but can also arise in machine deployment where the results from code can diverge when hardware and software configurations are changed[21]. So even perfection in one's own software environment does not guarantee reproducibility. As a result, for true reproducibility and consistency not only would we urge code release, but also a description of the hardware and software environment that the program was executed and developed.

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, data and 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 an assumption that program descriptions are adequate (something we address in this article); and the fact that the development of program code is a subsidiary activity in the scientific effort.

A modest proposal

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

- **SourceCode-Full**: Full release of *all* source code used to produce the published results along with the make recipes to recreate the executables and a series of regression tests to build confidence. (Anybody who has used Perl modules from CPAN (http://cpan.org, accessed 27-09-2011) will understand the value of this).
- **SourceCode-Partial**: Full release of source code written by the researcher accompanied by associated documentation of ancillary packages used, for example commercial scientific subroutine libraries.
- **SourceCode-Marginal**: Release of executable code and an application programming interface to allow other researchers to write test cases.
- **SourceCode-None**. No code provided.

This would would alert both the readers and authors of a journal article to the fact that the issue is important and it would also serve to 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 such as a patent or some copyright then there is a problem. It is difficult to see how to deal with this within the financial limitations of journals. Perhaps the simplest solution is for a journal to flag the software as “SourceCode-None” until such time as they can be reproduced, either because the software goes into the public domain or is released under some free licence. No slight is meant here. It simply says that for the moment, the results are not currently reproducible.

Limited access

Researchers may not have access to at least some of the software packages that are used for development. This would, we suggest, not be a problem for most researchers: their institutions would normally provide such software. If it was a problem then a journal could mark a publication as "SourceCode-Partial". The release would still be valuable in that it would address issues such as dissection detailed in the first part of our article 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 whilst 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 describe each of the released modular components of any software used and that editors and reviewers be empowered to include an appraisal of this in their judgment about the publication potential of the article. A good example of this is the way that the journal *Geoscientific Model Development* asks authors to describe their program code.

Logistics

The logistics of releasing and storing code whilst maintaining a cooperative development environment have been solved by the open source community in the last two decades. SourceForge (http://www.sourceforge.net/, accessed 26-09-2011) is an excellent exemplar and already home to numerous scientific packages. We would 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 system Latex, and *GenePattern-Word RRS* a system specific to genomic research[34].

It is still early days however. There are some projects that are exemplars, for example Donoho and co-workers at Stanford University have developed software packages that allow anyone with access to the *Matlab* system to reproduce figures from their harmonic analysis articles, inspect source code, change parameters and access datasets[36].

A Pathway to Implementation

Our thesis is that previous journal and funding body strictures relating to software implementations of scientific ideas that have held in the 20th century have become obsolete.

We have suggested one modest solution to code availability in this article. There are a number of further steps that journals, academies and educational organisations might consider taking:

- Research funding bodies should commission research and development on tools that enable code to be integrated with other elements of some scientific research such as data, graphical displays and the text of an article.
- Research funding bodies should provide meta-data repositories that describes both program and data produced by researchers. The Australian National Data Service which acts as an index to data held by Australian research organisations is one example of this
- Journals should expect researchers to provide some modular description of the components of the software that supports a research result; referees should be empowered to include an appraisal of this as part of their reviewing task. An example of this can be seen in a recent article published in *Geoscientific Model Development*[36].
- Science departments should expand their educational activities into reproducibility. Clearly such departments should, in the main, be teaching material that relevant to the science; however, courses on statistics, programming and experimental method could be easily expanded and combined to form offerings that have reproducibility at their heart.

Acknowledgements

We would like to thank David Hales of the Department of Design at the Open University and Paul Piwak of the Department of Computing for pointing out some reproducibility work outside computing. JGC is grateful to Ilya Goz for pointing out that there appeared to be calculation errors in the CRUTEM data from the Meteorological Office

Author Contribution

DCI, LH and J G-C contributed to all aspects of this article.

References

1 Barnes, S. Publish your computer code: it is good enough. *Nature* **467**, 753-753 (2010).

2 McCafferty, D. Should code be released? *Comm. ACM* **53**, 16-17 (2010).

3 Merali, Z. Computational science:..error. *Nature* **467**, 775-777 (2010).

4 Hanson, B., Sugden, A. & Alberts, B. Making data maximally available. *Science* **331**, 649-649 (2011).

5 Peng, R.D. Reproducible research and biostatistics. *Biostatistics* **10**, 405-408 (2009).

This work provides a succinct and convincing argument for reproducibility. *Biostatistics* are at the forefront of ensuring that code and data are provided for other researchers.

6 Editorial. Devil in the details. *Nature* **470**, 305-306 (2011).

7 Pederson, T. Empiricism is not a matter of faith. *Computational Linguistics* **34**, 465-470 (2008).

8 Donoho, D.L. An invitation to reproducible computational research. *Biostatistics* **11**, 385-388 (2010).

9 Raymond, E.S. *The Cathedral and the Bazaar*, Sebastopol, CA: O'Reilly (2001).

10 He, Y. & Ding, C.H.Q. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *Journl. of Supercomputing* **18**, 259-277 (2001).

11 Hatton, L., Wright, A., Smith, S., Parkes, G., Bennett, P. & Laws, R. The seismic kernel system-a large scale exercise in fortran 77 portability. *Software Practice and Experience* **18**, 301-329 (1988).

12 Hatton, L. The T experiments: errors in scientific software. *IEEE Computational Science and Engineering* **4**, 27-38 (1997).

13 Hey, T., Tansley, S. & Tolle, K. (Eds) *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Seattle, WA: Microsoft Press (2009).

14 Easterbrook, S.M. & Johns, T.C. Engineering the software for understanding climate change. *Computing in Science and Engineering* **11**, 65-74 (2009).

15 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. *Journl. Geophysical Research* **111**, doi:10.1029 (2006).

16 Adams, E.N.: Optimizing preventive service of software products. *IBM Journal of Research and Development* **28**, 2-14 (1984)

17 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. Berlin: Springer-Verlag (2010).

18 Yang, H., Willis, A., De Roeck, A. & Nuseibeh, B. Automatic detection of nocuous coordination ambiguities in natural language requirements. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, 53-62 (2010).

19 de Bruijn, F. and Dekkers, H. Ambiguity in natural language software requirements: a case study. In *Requirements Engineering: Foundation for Software Quality* (Eds. Wieringa, R & Persson, A.) 233-247. Berlin: Springer-Verlag (2010).

20 van Deemter, K. *Not Exactly*. Oxford: Oxford University Press (2010).

21 Monniaux, D. The pitfalls of verifying floating-point computations. *ACM Trans. on Programming Languages and Systems* **30**, 12:1-12:41 (2008).

22 Pfleeger, S.L. & Hatton, L. Investigating the influence of formal methods. *IEEE Computer* **30**, 33-43 (1997).

23 van der Meulen, M. J. P. and Revilla, M.A. The effectiveness of diversity in a large population of programs. *IEEE Trans. Software Engineering* **34**, 753-764 (2008).

24 Revol, N. Standardized interval arithmetic and interval arithmetic used in libraries. *Proc. 3rd Intl. Congress on Mathematical Software*, 337-341 (2010).

25 Rump, S.M., Ogita, T. & Oishi, S. Accurate floating point summation part 1: faithful rounding. *SIAM Jnl. of Scientific Computing* **31**, 189-224 (2008).

26 Rump, S. M. Accurate and reliable computing in floating-point arithmetic. *Proc. 3rd International Congress on Mathematical Software*, 105-108 (2010).

27 Badin, M., Bic, L., Dillencourt, M. & Nicolau. A. Improving accuracy for matrix multiplications on GPUs. *Scientific Programming* **19**, 3-11 (2011).

28 Boehm, B, Rombach, H.D and Zelkowitz M.V. (Eds) *Foundations of Empirical Software Engineering: the Legacy of Victor R. Basili*, Berlin: Springer (2005).

29 Thomas, S.J., Hacker, J.P., Desagne, M. & Stull, R.B. An ensemble analysis of forecast errors related to floating point performance. *Weather and Forecasting* **17**, 898-906 (2002).

30 Pan, V. Y., Murphy, B., Qian, G., & Rosholt, R.E. A new error-free floating-point summation algorithm. *Computers and Mathematics with Applications* **57**, 560-564 (2009).

31 Boldo, S & Muller, J-M. Exact and approximated error of the FMA. *IEEE Trans. Computers.* **60**, 157-164 (2011).

32 Kahan, W. Desperately needed remedies for the undebuggability of large floating-point computations in science and engineering, IFIP/SIAM /NIST Working Conference on Uncertainty Quantification in Scientific Computing.

33 Hatton, L. & Roberts, A. How accurate is scientific software? *IEEE Trans. Software Engineering* **20**, 785-797 (1994).

34 Mesirov, J.P. Accessible reproducible research. *Science* **327**, 415-416 (2010).

35 Donoho, D.L., Maleki, A., Rahman, I.U., Shahram, M. & Stodden, V. Reproducible research in computational harmonic analysis. *Computing in Science and Engineering* **11**, 8-18 (2009).

Donoho and fellow researchers have been at the forefront of reproducibility for many years. This article describes their work and their experiences, particularly with respect to tools for code presentation. Note that this journal renumbers pages for each issue, the issue number is 1.

36 Yool, A. , Popova, E.E. & Anderson, T.R. MEDUSA-1.0: a new intermediate complexity plankton ecosystem model for the global domain. *Geoscientific Model Development* **4**. 381-417 (2011).

An example of an article from a journal that proactively encourages code release. Appendix B of the article provides a description of the software.

Methods

Supplementary information including the software that was used to check the meteorological database can be found linked to the online version of this article.