Managing Advanced Computational Resources to Encourage Best Practices for Developing Repeatable Scientific Software

Dirk Colbry
Michigan State University
567 Wilson Road, Room 1440
East Lansing, MI
(517) 355-0730
colbrydi@msu.edu

ABSTRACT

Many researchers are under the impression that computers and software are perfect black boxes: you put in inputs and get an output, and the same input always produces the same output. However, experienced researchers and software developers have learned that the mapping of inputs to outputs is not always exact. In order to develop software that allows researchers reproduce the same – or at least scientifically consistent – results, it is important to understand the factors that cause computational variation. This paper explores some of the common sources of computational variation and discusses approaches that managers of advanced computing resources can use to help developers design robust scientific software that produces reproducible results.

Categories and Subject Descriptors
K.6.3 Software Management (D.2.9)

General Terms
Design, Reliability

Keywords
Reproducible Science, Software Development, High Performance Computing

1. INTRODUCTION

Experimental reproducibility is a foundation of the scientific method, as scientific knowledge and approaches cannot be used unless they provide predictable results. Computers are generally believed to produce exact calculations and accurate, precise results – yet in practice, complex computational systems can produce large variations in scientific results. Careful management of computational systems and software can support the production of more consistent results, and at Michigan State University this management role is fulfilled by the Institute for Cyber-Enabled Research (iCER) and the High Performance Computing Center (HPCC). Together, iCER and HPCC staff educate users about systems and processes, and encourage the development of software that produces accurate, reproducible scientific results.

As an example of the challenge of maintain complex computational resources, consider the following complaint that is frequently emailed to HPCC staff at MSU: “My code runs fine on my desktop, but breaks when I run on the HPCC. Please fix your computer system.” Our staff has learned from experience that the first step in resolving these researchers’ concerns is to understand (and manage) their assumptions about computational systems.

Most users are unaware of the many factors that impact the functionality of software – and that can ultimately influence scientific results and reproducibility. Some of the most common variables include: hardware type and configuration; operating systems; compilers; libraries; input parameters; and datasets. Without an understanding of these factors, users often assume that their software “breaks” because of some fundamental problem in the HPCC system. While this is sometimes the case, more often the problem is rooted in the design of the software itself or in the users’ limited understanding of the interrelationships between the code they write and the systems, libraries, and tools used to interpret that code on various computational platforms.

Managing user assumptions is difficult, and the MSU iCER and HPCC staff work hard to balance the need to produce accurate results in a timely manner with the goal of developing robust software that supports repeatable scientific inquiry. To address this challenge, iCER takes a two-fold approach: (1) developing a broad infrastructure of advanced computer hardware, software, and support resources – including multiple versions, obscure libraries and legacy codes – to support ongoing scientific discovery; and (2) providing education and support to help researchers understand good scientific software practices and develop robust programs that provide accurate, repeatable results.

The next section offers some best practices for developing software that supports reproducible science. Section 3 provides a more detailed example of scientific repeatability in the context of random number generation. Section 4 describes the efforts of iCER and the HPCC to support these best software practices, and Section 5 concludes with a discussion.

2. SOFTWARE BEST PRACTICES

Although highly skilled in their own disciplines, many scientists who develop research software are not formally trained in software design and development. Certainly, formal training is not required in order to write robust and high quality software; however, the time pressures of graduate programs and grant cycles commonly result in “hacked together” code that produces just enough data for the scientist or student to reach the next milestone (paper, thesis, grant proposal, etc.). Consequently, many research software tools will only work with very specific configurations of hardware/software/libraries.
Another challenge is that software developed within a research group is frequently “handed down” from student to student, with each subsequent author adjusting the original code to suit a new experiment or configuration. This frequently results in software that “mostly” works, but may produce irregular results or unpredictable behaviors – and brings researchers to iCER with pleas to “fix it” just enough to generate the data needed for their next deadline.

During these types of code review, it is common for the scientist-programmers to believe they have built the “Taj Mahal” of software: the very best approach and tool for doing science in their field. Yet, when the code is examined from a software design point of view its structure looks more like a shack (Figure 1): a mismatch of approaches, techniques and tools cobbled together over time to suit immediate needs, with little or no consideration for the overall structure, usability or flexibility of the code. Certainly, this difference in perspective leads to challenges. Scientist-programmers often come to iCER and HPCC looking for a quick fix to a specific problem that is impeding research progress, yet the experienced programmers often see that adjusting one aspect of the poorly-constructed code may have unanticipated consequences elsewhere.

The remainder of this section describes some of the “best practices” for researchers engaged in scientific software development. Following these approaches can help scientists develop more robust code that produces accurate, repeatable results, and avoid many of the problems we frequently encounter at iCER with poorly constructed, legacy codes.

![Figure 1: (a) How many researchers see their code. (b) How professional software developers see the same code.](image)

2.1 Good Software Carpentry

Software Carpentry [1] is an approach to educate and use tools and techniques that will help researchers get more done in less time. While there are many techniques in the Software Carpentry movement, the tenets most applicable to developing research software include the following:

- **Make all scientific software open source**: “Open Source” software is software that includes the source code to allow individuals to see how the software works. Open source does not mean (non-profit): there are many successful, for profit, scientific software companies that embrace open source philosophes [2]. The open source movement is a natural fit with the goals of reproducible science, which requires that scientific code be transparent so it may be scrutinized and experimental results can be replicated. This transparency is key for scientists to understand what has been done in the past; vet and fix problems; and allow future researchers to build off their work.

- **Use version control systems**: Version control systems allow software developers to manage their software by tracking changes and enabling collaborators to work together efficiently. When used correctly, distributed version control systems (such as Git [3] and Mercurial [4]) provide change history, track the evolution of software, make it easier to track down bugs, and allows researchers to work collaboratively on a common code base.

- **Use standard build systems**: Build systems compile, link and install software on a target system. The best approach is to use the build system designed for the programming language and hardware architecture. For example, makefiles, cmake files, and automate files are great solutions. Cmake is particularly good if the scientific code is to be made cross platform [5]. For Python, an effective approach is to wrap code using easy_install and pip. When it is not possible to use one of the standard build systems, custom scripts or README files should be designed to use one command or a minimal, standard set of commands; this approach allows new users to easily install and run the code. Software that relies on heavily customized scripts or complex installation processes generally result in difficult builds and are extremely hard to debug when problems arise.

- **Use standard testing systems**: For many scientist-programmers, testing the code is limited to ensuring that the correct results are produced at the end of a simulation or calculation. However, this is not a robust approach. Instead, unit testing involves testing software at the lowest logical level. Testing systems should be designed for the target platform and programming language, and ideally the unit testing should be wrapped with the build system. This allows individuals to install and test the software, even if they do not understand the underlying purpose and science of the software. For example ctest works with cmake and Sphix (or equivalent) for Python [6].

- **Use consistent coding and documentation standards**: All software languages have code standards (some have many); following a single, consistent standard within a program makes the code much easier for others to interpret. Auto documentation standards can be particularly helpful, such as javadoc (java), doxygen (C, C++, and others) and pep8 (Python). These standards encourage good commenting practices while writing the code, and then help researchers generate the documentation automatically.

2.2 Test Software Robustness

Software is “robust” when it produces the desired results even when varying the inputs or running environment. Having robust code helps make science reproducible, as system environments change frequently. The following is a list of tests that can help ensure software will produce reliable and repeatable results.

- **Test software using different hardware**: Researchers should test their code on a variety of hardware and ensure that answers are consistent. Different hardware (ex: Intel vs. AMD) should not affect the results of a robust program; however, it is important to test this and not simply assume that the code will work on different platforms. This is especially true for parallel code when communication fabrics and processor speeds can significantly change the order of communication.

- **Test software using different libraries**: When incorporating standard libraries or interfaces in research software, it is important to test the program with different libraries and ensure the answers are consistent. For example, many math libraries perform the same functions and are, in theory, interchangeable (e.g., lapack, flt, Intel Math Kernel Library). Another good
example is the Message Passing Interface (MPI); in theory, most code written using OpenMPI should link fine to MPICH and MPICH2. However, it is important to test software across these different implementations and libraries to ensure the code is robust and portable.

- **Test software using different compilers:** Compilers such as C, C++ and Fortran all follow an agreed upon standard and, in theory, all should produce the same results. However, in reality different compilers do things in different ways and this can cause variations in output. This is especially true when the language standards are not well defined; for instance, one implementation of C may interpret the standard differently than another implementation. Generally, researchers should stick to the code standard and avoid using “tricks” that work only for one compiler implementation, as this makes their code non-standard and less portable. Another benefit of compiling and testing code on multiple implementations is that different compilers may provide different warnings and error messages, which can be useful in tracking down bugs in the code.

- **Test software using a variety of parameters and input data:** Obviously varying the input to scientific code causes the output to vary. However, many input parameters are estimated. It can be helpful to test code by making small changes to these estimates and ensuring that only small changes are made in the output. Chaos in scientific codes is well documented [7], and it is important to identify how code varies with input parameters.

## 2.3 Publishing All Required Parameters

In order to ensure reproducibility, many fields require that publications include “Methods” sections where researchers detail the steps that are taken to produce the results. However, publications in computational science often do not provide this level of detail. Instead, algorithms are often described at a high level and results are summarized in graphs. This lack of detail makes it difficult or impossible to reproduce the method and results.

Instead, publications of computational science research should include an equivalent “methods” section that reports the parameters used to achieve the presented results. More specifically, consider reporting the following:

- **Hardware:** the number, type and speed of processors should be documented, along with memory and the types of communication hardware and protocols involved. This information is particularly important when the scientific results document the speed of calculations, which can be highly influenced by processing, memory and communications hardware.

- **Compiler:** Different compilers (or different versions of the same compiler) can produce different results. Thus, it is important to report the compiler version and parameters.

- **Operating System:** Providing information on the operating system may be helpful for certain experiments, particularly if they are likely to be reproduced by others.

- **Linked libraries:** Researchers should report libraries that are linked as well as their version numbers. These details are especially important in library-rich languages like Python; using standard projects such as Anaconda [8] and Enthought Python [9] can also help others reproduce results.

- **Software version information:** Scientific codes are constantly evolving, and good version control systems are used to track these changes. In many cases, different versions of the code may produce different results so it is important to report the version control commit hashes used for specific experiments. This is the approach used in the ENZO (Hydrodynamic code) and YT (Visualization) projects [10], which then allow subsequent researchers to reproduce the same version and configuration of the code.

- **Input parameters and data:** Many programs use adjustable variables to set up the computation, and researchers frequently set these variables to “Magic Numbers” that they have found will produce the desired results. Clearly documenting these variables and settings is important to ensure future reproducibility.

## 3. EXAMPLE: RANDOM NUMBERS

Consider the special case of using pseudo-random numbers in scientific code. Random number generation is central to many computational science approaches, such as Monte Carlo methods. However, the nature of random number generation can lead to variability in the results without careful documentation. Consider the scenarios described below.

**Statistically Consistent Results:** When random numbers are used, it can be difficult to provide exactly reproducible results. If this variation is unavoidable, then researchers should run enough experiments in order to ensure robust, statistically consistent results. This is a common approach in natural science, as scientific variation in the real world is the norm and it would not be expected to have identical results for every scenario. However, it is important to run enough experiments to understand the range of appropriate results – and to identify inconsistencies that point to problems in the code or in the data.

**Random Seeds:** Most pseudo-random number generators allow for researchers to provide a random seed. In this case, using the same random seed will consistently produce the same series of random numbers; while a different random seed will produce different results. In order to ensure reproducibility, researchers should include the random number seeds that were used, or at least provide information about the number of seeds that were used to establish the statistical confidence of the results. Care should also be made to identify and report anomalous results.

**Random Number Libraries:** In addition to the random seed, the name and version number of the random number generation library used should be included. Altering the version or operating system may produce different results, even with the same random number library. In order to produce consistent random number generation across platforms, researchers may want to code their own random number generator (as was done with the Avida digital evolution project [11]).

**Parallel Random Numbers:** Random number generation becomes extremely difficult when working with parallel code [12]. Projects such as the C++ parallel random number generator library provide central control to ensure that with the same seed, the same quantity of random numbers are generated in the same order regardless of the number of processors and the communication link used [13].

## 4. ENCOURAGING BEST PRACTICES

At Michigan State University, iCER and the HPCC provide many resources to support researchers in using advanced computational tools to conduct science. As part of this support, we have designed the following programs and policies to help encourage best practices when developing scientific software:
• **Training:** iCER and HPCC offer extensive training opportunities, both in formal seminars and classes and through informal, one-on-one consulting appointments. Regular workshops are offered to introduce and reinforce Greg Wilson’s Software Carpentry initiative [1]. iCER research consultants also work individually with users to review code, suggest technical and structural improvements, and refer individual scientist-programmers to additional resources as appropriate.

• **Version Control Server:** All MSU faculty, staff and students have access to create both public and private repositories on our locally-hosted instance of Gitlab [14]. Faculty members are encouraged to integrate Gitlab into their classrooms, and special permissions can be set up to keep student projects private for grading purposes.

• **Extensive Software Stack:** With over 400 different software titles, the HPCC tries to install and maintain software that researchers need. In addition, HPCC offers a variety of software libraries and compilers, which allows researchers to easily test their software with different configurations. To support robust testing, libraries are compiled multiple times with multiple compilers and compiler versions. All of these packages are managed using the LMOD module system [15], which allows users to quickly switch compilers and libraries when running tests of software robustness.

• **Software Development Tools:** The HPCC also installs tools that enable good software development, such as installer frameworks (make, cmake); testing packages (ctest); and tools for profiling, delinting and debugging code. Training and support for using these development tools is provided by iCER research consultants, ensuring that users are able to take advantage of these resources.

• **Hardware:** The main HPCC compute cluster supports a variety of computing nodes including: large shared memory nodes (1-2TB and 64 cores); GPU accelerated nodes (M1060 and K20s); and Phi Accelerated nodes. In addition to the cluster, which is run using a batch scheduler, the HPCC also provides a variety of developer and education nodes specifically designed for user testing and benchmarking. Retired hardware is made available for testing reverse compatibility, as well as newer, cutting-edge hardware that is not yet available in the main cluster. Some examples of these specialty nodes include; Adam nodes, PlayStation Cell Processor, GPU cards with K10 and K40s and an old Intel Itanium node.

• **Tool Development:** iCER staff works directly with researchers to develop tools to facilitate best practices. For example, sys-inspector ([https://gitlab.msu.edu/becketta/sys-inspector.git](https://gitlab.msu.edu/becketta/sys-inspector.git)) is an easily extensible Python tool that reports information about the Python environment used to run experiments, including: operating system, Python version, installed Python modules, and git hash of project (if applicable). Another example is the cpp_params class source files ([https://gitlab.msu.edu/kineticsn/cpp_params.git](https://gitlab.msu.edu/kineticsn/cpp_params.git)) that enables users to easily flag input parameters inside their code and automatically builds a command line interface that can read in parameter files, write them out, and adjust program parameters on the command line when the program starts; this makes it easy for users to run parameter sweeps, test their code with a variety of inputs and report the results.

5. **CONCLUDING DISCUSSION**

In order to be truly valuable, good science must be reproducible. This reproducibility can be difficult to achieve when computational science software is developed as a “black box” that hides its function and parameters, or when hardware and software differences introduce variations in the results.

To improve transparency and ensure that their science is reproducible, scientists should develop programs that follow programming standards; should test for robustness across platforms and configurations; and should use software carpentry tools to help ensure, robust reproducible code. When reporting results, papers should be written with comprehensive “methods” section to give fellow scientists the best chance for producing the same results. In addition, managers of advanced computing hardware can help support scientist-programmers by providing training, maintaining comprehensive software stacks (including variety of compilers, libraries, and programming tools) and providing expertise to assist researchers in their software development.

6. **ACKNOWLEDGMENTS**

Special thanks to Aaron Beckett for his significant contribution to the development of system-inspector. Additional thanks to Jim Howell and Alec Staber for their use and testing of cpp_params.

7. **REFERENCES**