Dissemination and Management of Computational Science Software

Matthew G. Knepley
Computation Institute
University of Chicago, Chicago, IL
knepley@ci.uchicago.edu

1. Barriers to Sharing

Today large, sophisticated, parallel codes have been used to simulate problems as diverse as supernovae, protein binding, earthquakes, and fluidized bed reactors. However, these breakthrough computational results and packages are all but unavailable to other scientists, greatly limiting the impact of these experiments.

Many codes are *open source*, in the sense that a user might obtain a raw copy of the code. However, this is far from sufficient if one is to reproduce the computation itself. *Transparency* must extend to the installation process, verification tests, benchmark computations, and analysis of output. Moreover, these large codes are frequently dependent on many smaller packages. This transparency must extend down the entire hierarchy of packages.

In the same way, the ability to *repeat* a given calculation is a foundational step, but ultimately insufficient. Another user must be able to easily alter parameters, introduce new experiments, and interact with the output. The code must be *usable* by others.

In the next sections, we will attempt to describe a set of necessary tools for sharing scientific code and reproducing computational experiments. Note that most of the considerations here also apply to publication of computational results.

2. Location and Retrieval

2.1. Retrieval

The most common method of disseminating source code today is to make a tar archive of the source tree available for download on the web. This scheme, however, has several limitations. It can be difficult to determine exactly which archive we received. This problem is usually corrected haphazardly with descriptive file names, or in a more coherent fashion using digital signatures. An elegant solution to this problem is provided by *version control* systems.

A version control system (VC) tracks changes made to files under its management. Older systems, such as CVS and Subversion, track all changes individually, whereas more modern systems, such as Mercurial and Git allow sets of changes to be grouped together. Thus, using version control, a user is freed from the dependence on a particular release, and can choose to obtain a snapshot of the software at any point in its development. The structure of releases is recovered by labeling particular changes with release information.

VC also gracefully handles the related problem of inevitable bug fixes and updates, usually refered to as *change management*. Entirely new archives are expensive to download, and patches are a cumbersome, error prone mechanism. VC allows the user to seamlessly retrieve updates, and some systems such as Git or Mercurial Patch Queues, allow them to select only the updates they want. Maintaining several versions or reverting to a previous version, quite difficult with different tarballs, can be easily done as well.

Finally, with the introduction of VC, the situation for potential contributors changes dramatically. Instead of submitting a *diff*, or text description of the changes made to some version of the code, the VC system can send a structured description not only of the changes, but the base version which was changed. Since the introduction of Git, contributions to the Linux kernel have drastically jumped [1]. Moreover, distributed VC systems such as Mercurial and Git give the user a full record of past changes to explore, and allow users to exchange updates directly, without the intervention of a central server.

2.2. Location

A user must also locate the package he is searching for. There are several excellent free hosting services, including SourceForge, BitBucket, GoogleCode, and GitHub. These websites provide not only VC, but also issue

tracking, source code browsing, and a Wiki for user interaction. Inexplicably, however, these sites omit one of the most important resources for scientific software, namely published papers.

The arXiv preprint server [2], operated by Cornell University, is the free source for papers on scientific computing. However, it does not currently support submission of associated source code. The arXiv does incorporate version control, however it does not leverage any existing package, and omits many features necessary for source code [3]. In addition, this software does not appear to be open, an ironic situation for one of the largest open archives of scientific papers. It seems clear that a merger of the functionality from the arXiv and a hosting site above would result in significant benefit for the computational science community.

3. Configuration, Build, and Run Management

3.1. Configuration and Build

The original aim of a configuration process was to preserve the *portability* of a piece of software. Tests would be performed to determine, for instance:

- which version control systems are available
- which languages have compilers or interpreters available
- which compilers should be used
- which compiler flags should be used
- what capabilities the compilers possess
- whether a data type is defined
- whether a header file is present
- whether a library is present
- whether a function is present in a library
- whether a executable file is present

This information is used to customize the build by generating headers, and makefiles or build scripts. This was the original motivation for the very popular GNU autoconf program, and is sufficient for the development of UNIX tools for which it was created.

As scientific software has evolved to solve much more complicated problems using advanced numerical methods, packages have evolved from self-contained libraries, such as BLAS/LAPACK, to frameworks which are built upon many lower level libraries, such as PETSc [4] which has interfaces to more than 60 packages [5]. Autoconf was not designed to handle a hierarchy of packages, and this rapidly becomes unmanageable. For PETSc, we have written a replacement in Python, BuildSystem, which is explicitly designed to interact with subsidiary packages. In fact, most packages to which PETSc interfaces can be automatically downloaded and built by the configure system. This style of package management is another implementation of the Gentoo Linux Portage philosophy.

In the experience of the author, the problem of subpackage management is a severe obstacle to sharing large scientific codes. Most codes do not possess a sophisticated configuration system, and are built on a small number of systems through great effort by the dedicated developers. Adoption of modern configuration tools is absolutely crucial to widespread dissemination of the large scale, complex frameworks on which modern simulations will depend.

Once the configuration process has customized build information for the target architecture, operating system, external package selection, compiler suite, and optimization level, the build process itself is quite straightforward. GNU make is by far the most popular software for build management, but recently competitors such as SCons and Jam have arisen which provide superior functionality at the cost of increased complexity.

3.2. Testing

Once installed, it is imperative to check the package operation, even cursorily. Construction of an executable can reveal unresolved symbols which would not cause a build error when using shared libraries. Also when using shared libraries, it can reveal mismatches between relocatable (produced with `-fPIC`) and non-relocatable libraries, the source of many errors on 64-bit platforms. For more sophisticated unit testing, we recommend using a framework, such as cppUnit. Unit tests are small pieces of code, having well defined output, which test a single capability of the program. These allow

the user to verify that installation has proceeded correctly and the code is producing correct results. A neglected part of unit testing, especially important for scientific codes, is performance tests. These can be validated against models of the performance parametrized by easily measured machine characteristics. This kind of modeling and testing is crucial for obtaining efficiency across a range of architectures. Finally, regression tests can be useful for longer term users to verify the installation after updates. Regression tests are larger pieces of code, testing multiple functionalities at once. The test checks for changes in the result, rather than an analytically known answer. Excellent free systems for this are available, such as buildbot.

3.3. Run verification and code assessment

Once a package has been successfully installed and tested, the user can attempt to replicate computational experiments. Each package will inevitably have some idiosyncratic input. However, some tools have been developed which attempt to organize input, output, and analysis, and create common patterns and infrastructure for code use. These will be crucial if we are to progress from mere repeatibility of computational experiments, to usuability of packages, allowing others to push experiments beyond the original conceptions.

The Pyre framework from Caltech has developed a general system to encode input data using Python. A particularly useful feature of this new system is the flexible expression of units. All units are expressed as a tuple of the basic SI units, which is transparent to the users. Particular units are defined as static tuples in Python and all conversion can take place automatically. Pyre also provides a uniform interface for launching jobs on diverse architectures with different run management policies. This can be invaluable when trying to reproduce a large run made on an unfamiliar batch queueing system. This system is in use for a DOE ASCI project at Caltech.

Once an experiment, or benchmark, is executed, users must be able to interpret the result. Often these checks are quite simplistic, such as checking that a field is reproduced to a norm-wise tolerance, or that a residual norm is sufficiently small. However, more structured output and tests allow richer forms of interaction by the user.

A widespread method for comparing and understanding solution output is to visualize the result. Common formats, such as VTK, are now widely supported and complex visualizations can be easily constructed using free platforms, such as Mayavi2 and Paraview. However, visualization formats

like VTK discard much of the structure in the solution, and complicate further analysis. More advanced, but also more narrow, formats such as CGNS and ExodusII carry explicit information about the discretization method and physics of the problem. This information can be used to analyze the solution in ways unavailable to traditional visualization tools. The Cigma [6] tool from Caltech allows comparison of finite element solutions from different meshes and elements by explicitly projecting each to a canonical mesh and then evaluating error norms and visualizing differences using VTK. This kind of tool allows the user to connect directly with the FEM theory when evaluating a solution, rather than merely a picture.

4. Outlook

In order to fully realize the benefits of open code frameworks and reproducible computational experiments, the tools highlighted above must be combined into a single workflow. Usability is the key requirement for community adoption. It would seem natural to try and layer this functionality on top of existing systems. For instance, the arXiv is a natural candidate. We should also consider existing packaging systems, such as Apt and Yum. However, most important for success is a community commitment, from both developers and users, to hold computational experiments to the same high standard as those done in any laboratory.

References

[1] G. K. Hartman, The linux kernel (2009).
URL http://www.youtube.com/watch?v=L2SED6sewRw

[2] P. Ginsparg, Creating a global knowledge network, second Joint ICSU Press–UNESCO Expert Conference on Electronic Publishing in Science (February 2001).
URL http://people.ccmr.cornell.edu/~ginsparg/blurb/pg01unesco.html

[3] R. E. Luce, E-prints intersect the digital library: Inside the los alamos arxiv, Issues in Science and Technology Librarianship (29).
URL http://www.library.ucsb.edu/istl/01-winter/article3.html

[4] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory (2008).

[5] B. Smith et al., External Software Used by PETSc, http://www.mcs.anl.gov/petsc/petsc-as/miscellaneous/external.html.

[6] L. Armendariz, S. Kientz, Cigma user manual (2009).
URL http://www.geodynamics.org/cig/software/packages/cs/cigma