Python Tools for Reproducible Research on Hyperbolic Problems

Reproducible research in computational science is only possible if scientists distribute the computer codes they used to generate their published results or archive them in such a way that other researchers can later examine them. The author describes some difficulties in achieving this goal, along with a set of Python tools for facilitating reproducible research on finite volume methods.

Within the world of science, computation is now rightly seen as the third vertex of a triangle, complementing observation and theory. However, it has yet to reach the maturity of the experimental sciences in terms of reproducibility. Nowhere else in science can someone so easily publish observations that claim to prove a theory or illustrate a technique's success without giving a careful description of the methods used in sufficient detail so that others can attempt to repeat the experiment. In most branches of science, it's not only expected that publications contain such details, it's also standard practice for other labs to attempt to repeat important experiments soon after they're published. Although this might not lead to significant new publications, it's viewed as a valuable piece of scholarship and a necessary component of the scientific method.

Scientific and mathematical journals are filled with pretty pictures of computational experiments that the reader has no hope of repeating. Even brilliant and well-intentioned computational scientists often do a poor job of presenting their work in a reproducible manner. They often define their methods vaguely, but even if the methods are carefully specified, the reader would have to implement them from scratch to test them. Most modern algorithms are so complicated that there's little hope of doing this properly. Many computer codes have evolved over time to the point where even the person running a program and publishing the results knows little about some of the choices made during the implementation. And such poor records are typically kept of exactly which version of the code or parameter values were used that even a paper's author can find it impossible to reproduce the published results at a later time. Regrettably, I speak from ample first-hand experience here.

As Jonathan Buckheit and David Donoho point out in their classic paper on reproducible research (see www-stat.stanford.edu/~donoho/Reports/1995/wavelab.pdf), the scientific method and style of presenting experiments in publications that we currently take for granted in the experimental sciences were uncommon before the mid 1800s. Today, they're a required aspect of respectable research, and experimentalists are expected to spend a fair amount of time keeping careful lab books, fully documenting each experiment, and writing their papers to include the

1521-9615/09/$25.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Randall J. LeVeque
University of Washington

details needed to repeat experiments. The computational sciences might need a paradigm shift of the same nature.

The idea of "reproducible research" in scientific computing is to archive and make publicly available all the codes used to create a paper's figures or tables, preferably in such a manner that readers can download the codes and run them to reproduce the results. As Buckheit and Donoho put it, "An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures." They present this as a distillation of the insights of Jon Claerbout, an exploration geophysicist who has been a pioneer in this direction since the early 1990s.[1]

The development of very high-level programming languages has made it easier to share codes and generate reproducible research. Historically, many papers and textbooks contained pseudocode, a high-level description of an algorithm intended to clearly explain how it works but that won't run directly on a computer. Today, we can write many algorithms in languages such as Matlab or Python in a way that's both easy for the reader to comprehend and is fully executable, with all details intact.[2] In this article, I survey a set of Python tools for facilitating reproducible research on finite volume methods for hyperbolic conservation laws using the Clawpack software.

Objections and Obstacles

A natural objection to making code freely available for reproduction is that it takes a lot of work to clean it up to the point where someone else can just use it, let alone read it. Although this is certainly true, it's still well worth doing, not only in the interest of good science but also for the selfish reason of being able to figure out later what you did and build on it further.

Those of us in academia should get in the habit of teaching good programming, documentation, and record-keeping practices to our students and then demand it of them. We owe it to them to teach a set of computational science skills that will certainly become increasingly necessary in academic research environments and that are already highly valued in industrial and government labs. Learning this skill will also improve their chances of building on their own work after they graduate and of future students being able to use their contributions rather than starting from scratch, as is too often the case today.

Although ideally all published programs would be nicely structured and easily readable with ample comments, as a first step, it would help simply to provide and archive the working code that produced the results described in a paper. Even this takes more effort than you might think, though. It's important to begin expecting this as a natural part of the process so that researchers feel less like they have to make a choice between finishing off one project properly or going on to another where they can more rapidly produce additional publications. The current system strongly encourages the latter.

Requiring our students to do this might be a good place to start, provided we recognize how much time and effort it takes. Perhaps we should be more willing to accept an elegant and well-documented computer program as a substantial part of a thesis, for example. This isn't unreasonable—a thesis in mathematics, like a research paper in this field, typically contains long and detailed proofs of theorems that are unreadable to all but a handful of experts around the world. Many readers will be interested in the results without working through all the details, but the details should be provided. It's also expected that the student will spend considerable time perfecting these details and writing them up. Constructing a computer program isn't so different from constructing a formal proof.

A second objection to publishing computer code is that a working program for solving a scientific or engineering problem is a valuable piece of intellectual property, and there's no way to control its use by others once it's made publicly available. Of course, if the research goal is to develop general software, then it's desirable to have as many people using it as possible. However, for a scientist or mathematician primarily interested in studying some specific class of problems who developed a computer program as a tool for that purpose, there's little incentive to give this tool away free to other researchers. This is particularly true if the program has taken years to develop and provides a competitive edge that could potentially lead to several additional publications in the future. By making the program globally available once the first publication appears, other researchers can potentially skip years of work and start applying the program to their own problems immediately. In this sense, providing a program is fundamentally different than carefully describing an experiment's materials and techniques; it's more like inviting every scientist in the world to come use your carefully constructed lab apparatus free of charge.

This argument has considerable merit in some situations, but several counterarguments are also worth mentioning:

- In many cases, the code's development is supported by federal grants, so taxpayers have made the largest financial investment, not its authors. Accordingly, federal grant agencies increasingly demand that the work they support be made openly available. In fact, the US National Institutes of Health now require that papers containing work they fund must be made available via the National Library of Medicine's PubMed Central within 12 months of publication in a scientific journal.[3] Agencies that support code development take a similar attitude—a recent request for proposals from the US Department of Energy states that "successful applicants … must ensure that source code is fully and freely available for use and modification throughout the scientific computing community via a pre-approved open source process" (www.er.doe.gov/grants/FAPN06-04.html).
- It's notoriously difficult to take someone else's code and apply it to a slightly different problem, even when they collaborate and willingly provide hands-on assistance with the code's development. This is often true when the code's author describes it as general software that's easy to adapt to new problems, and it's particularly true if the code is obscurely written with few comments and the author isn't willing to help out, as would probably be the case of many research codes that people feel the strongest attachment to.
- My own experience in computational science is that virtually every computational experiment leads to more questions than answers. There's such a wealth of interesting phenomena that can be explored computationally these days that any worthwhile code can probably lead to more publications than its author can possibly produce. If other researchers can take the code and apply it in some direction that wouldn't otherwise be pursued, this should be seen as a positive development, both for science and the original authors, provided, of course, that they get some credit in the process. This is especially important for computational mathematicians, whose goals are often the development of a new algorithm rather than the solution of specific scientific problems. Even for those not interested in software development per se, anything we can do to make it easier for people to use the methods we invent will benefit our own careers.

Perhaps what's needed is an expanded copyright process for scientific codes, so that programs could be made available for inspection and independent execution to verify results but with the understanding that they can't be modified and used in new publications without the author's express permission for some period of years. This permission could be granted in return for coauthorship, for example. In fact, such a system already works quite well informally, and greater emphasis on reproducible research would make it function even better. It would be quite easy to determine when someone violates this code of ethics if everyone were expected to "publish" their code along with a paper. If the code is an unauthorized modification of someone else's, it would be difficult to hide.

A third obstacle to making programs freely available is that they're based on commercial, proprietary, or copyrighted software that can't be redistributed. This is certainly a limitation if the proprietary code is an integral part of the program, but in many cases it isn't. Often the part of the program that corresponds to the original research being published is the authors' work. Making it available could help readers understand the research even if they can't run it, and those who do have access to the proprietary code (such as by purchasing the required packages) can also run it. Certainly the large number of books and papers that include Matlab codes are valuable in spite of being written for a commercial platform that many readers can't afford.

Along the same lines, some codes only run on special hardware, such as massively parallel supercomputers that many readers might not have access to. But again, the ability to inspect the code and determine what parameter or algorithmic choices the authors made is often much more important than the ability to rerun the code and recreate the results already published.

Clawpack Software: A Case Study

The remainder of this article presents a brief case study to illustrate a set of tools that aid in the presentation of reproducible research on wave propagation algorithms for solving hyperbolic partial differential equations (PDEs). The reader need not be familiar with such problems to follow this discussion.

Hyperbolic PDEs model a variety of wave propagation and fluid flow problems, including acoustic and seismic waves, advective transport, shallow water theory, and compressible gas dynamics, to name just a few applications. In many cases, the equations are nonlinear systems of conservation laws whose solution contains shock waves or other discontinuities in the solution that are particularly difficult to capture with classical finite difference methods. Much of my work over the past 15 years has been devoted to trying to make it easier for myself, my students, and other researchers to perform computational scholarship in the development of numerical methods for hyperbolic PDEs and also in a variety of other application areas in science and engineering that use these methods. This work resulted in the Clawpack software (www.clawpack.org) and its extensions consisting of open source Fortran code that has been freely available since 1994. More than 7,000 users have registered to download this software since it first appeared and have used it on a wide variety of problems.

The details of the algorithms implemented aren't important for our purposes here. The interested reader can learn more about them in a textbook[4] that was designed to be used in conjunction with the software. Virtually all the figures in the book are reproducible in the sense that you can download the programs that generated them from a Web page and easily execute them. Most figure captions contain a link to the corresponding Web page with the code, along with additional material not in the book—for example, animations of the solution evolving in time. The problem-specific code for each example in the book is quite small and easy to comprehend and modify. The reader is encouraged to experiment with the programs and observe how changes in parameters or methods affect the results. These programs, along with others available at www.clawpack.org, can also form the basis for developing programs to solve similar problems.

Although Clawpack's development was originally motivated by the desire to make a set of existing methods more broadly accessible, the software's availability has also encouraged me to pursue new algorithmic advances that I otherwise might not have. I hope that it will also prove useful to others as a programming environment for developing and testing new algorithms, and for comparing different methods on the same problems. Because the source code is available and the basic Clawpack routines are reasonably simple and well documented, it should be easy for users to modify them and try out new ideas. I encourage such use—I certainly use it this way myself.

Careful direct comparisons of different methods on the same test problems are too seldom performed in the study of methods for hyperbolic problems, as in many computational fields. One reason for this is the difficulty of implementing other peoples' methods, so the typical paper contains only those results obtained with the authors' method. Sometimes (but not always), they test the method on standard problems and compare the results with others in the literature. Often the reader must be content with comparisons in the "eyeball norm" since many papers only contain plots of the computed solution and no quantitative results. Of course, many exceptions to this exist, including papers devoted to careful comparisons of different methods, but these are still a minority. I hope that Clawpack might facilitate this process more in the future and that other algorithms for hyperbolic PDEs might be provided in a Clawpack-style implementation that encourages direct use and comparison by others. One example in this direction is the WENOCLAW software developed primarily by David Ketcheson[5] that implements a class of higher-order methods.

Recently, my students and I have been developing a set of Python tools to facilitate the use of Clawpack. We have several goals for new features:

- *A wider range of graphics and visualization options for viewing results computed with this software.* Traditionally, users have relied on Matlab as the primary visualization tool, since we provided several Matlab scripts and functions as part of Clawpack for visualizing results. Although Matlab is familiar and convenient for many users, this commercial package isn't available to everyone, and we wish to provide open source alternatives. Moreover, in three dimensions, Matlab graphics aren't as powerful as other available packages, and in particular don't provide voxel graphics for volume rendering. We created the plots presented in this article and at www.clawpack.org/links/cise09 by using matplotlib in Python, available in SciPy (http://scipy.org) and sufficient for many 1D and 2D plotting purposes (3D data requires other packages). We're currently developing Python interfaces between Clawpack and other open source visualization tools, in particular the VisIt software at Lawrence Livermore Laboratory (www.llnl.gov/visit/), which supports a wide variety of tools for 2D and 3D on adaptively refined grids.
- *Literate programming tools for documenting code with mathematical expressions that are easy to read and that link to other parts of the code, external documentation, or Web pages with more information.* Donald Knuth[6] coined the term *literate programming*, which refers to programs that are fully self-documented in a manner that's human readable. Literate programming techniques can greatly assist in the development of reproducible research in computational science.

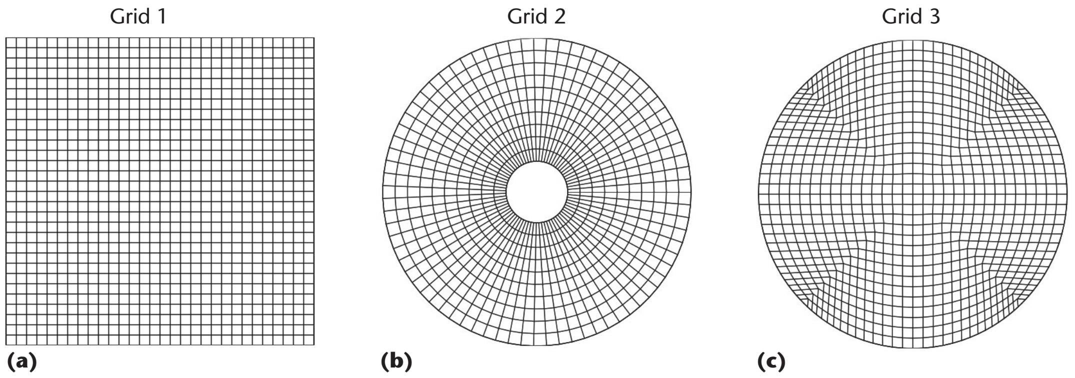

Figure 1. Three types of grids used for the advection test. Here, we use (a) a Cartesian grid, (b) polar coordinates, and (c) a nonstandard quadrilateral grid recently proposed[7] as an alternative.

- *A Web-based interface to Clawpack.* The goal is both to facilitate its use by students learning about hyperbolic problems (so they can easily experiment without having to work directly with the Fortran code and data files) and to have a Web portal so that these experiments didn't require downloading and installing the Clawpack software locally.
- *Templates for performing reproducible research.* Many of the same tools needed for the Web portal (EagleClaw) are also useful in developing scripts that run a series of tests and collecting the results in a Web page or LaTex documents.

We wrote most of the Clawpack software in Fortran 77. The code reads in a set of parameters from ASCII text files with names such as claw2ez.data (which contains parameters that define the computational domain, the number of grid cells, the method to be used, and so on) and setprob.data (which contains user-defined, problem-specific parameters). Running the code creates a set of output files with the solution at several user-defined output times. These files (either ASCII or hdf binary) are then read into a visualization tool (Matlab or Python). In designing a Python interface, we retained this basic structure and developed tools that interact with the Fortran code by modifying the data files, which have essentially the same form as in previous versions of Clawpack. This means that users can still work in the classical manner if they desire and need not use Python at all if they prefer to modify the data files by hand. Previous applications will continue to run unchanged.

To illustrate these tools and their use, let's look at the simplest possible hyperbolic equation in two space dimensions: the advection equation modeling the transport of a tracer in a specified velocity field. As a test problem, we consider "solid body rotation," in which the velocity field is

$$u(x; y) = -2\pi y, \qquad v(x; y) = 2\pi x,$$

corresponding to a counterclockwise rotation about the origin with period 1. The initial data we used is also at www.clawpack.org/links/cise09, along with figures and animations of the numerical solution and all the source code used to generate these results.

We compare the behavior of the algorithms implemented in Clawpack on three different types of computational grids, as illustrated in Figure 1: Cartesian grids with square grid cells (grid 1), polar grids in an annulus (grid 2), and quadrilateral grids (grid 3).[7]

For each grid type, we compute the solution at four different grid resolutions to estimate the method's order of accuracy. We also compare two different numerical methods. The first (with `Limiter = 0`) is a variant of the Lax-Wendroff method, which should be second-order accurate on smooth solutions and smooth grids but is dispersive and often produces nonphysical oscillations in the numerical solution. The second method is one of the high-resolution limiter techniques implemented in Clawpack (with `Limiter = 3`, a particular choice known as the monotonized centered limiter[4]). This method is no longer formally second-order accurate but often performs much better in practice, particularly on problems with discontinuous solutions such as the shock waves that often arise in solving nonlinear hyperbolic equations. Tables 1 through 3 display the results from these 24 test cases (four resolutions on each of three grid types, comparing two different nu-

Table 1. Errors on grid 1.

			Limiter = 0		Limiter = 3	
mx	*my*	Ave Δ*x*	Error	Observed order	Error	Observed order
30	30	0.0591	0.3263	nan	0.1490	nan
60	60	0.0295	0.1336	1.29	0.0404	1.88
120	120	0.0148	0.0389	1.78	0.0088	2.20
240	240	0.0074	0.0105	1.89	0.0020	2.14

Table 2. Errors on grid 2.

			Limiter = 0		Limiter = 3	
mx	*my*	Ave Δ*x*	Error	Observed order	Error	Observed order
10	75	0.0634	0.0917	nan	0.0408	nan
20	150	0.0317	0.0297	1.63	0.0112	1.86
40	300	0.0159	0.0084	1.82	0.0033	1.78
80	600	0.0079	0.0023	1.89	0.0008	1.97

Table 3. Errors on grid 3.

			Limiter = 0		Limiter = 3	
mx	*my*	Ave Δ*x*	Error	Observed order	Error	Observed order
30	30	0.0591	0.3843	nan	0.1610	nan
60	60	0.0295	0.1567	1.29	0.0492	1.71
120	120	0.0148	0.0456	1.78	0.0123	1.99
240	240	0.0074	0.0123	1.89	0.0034	1.86

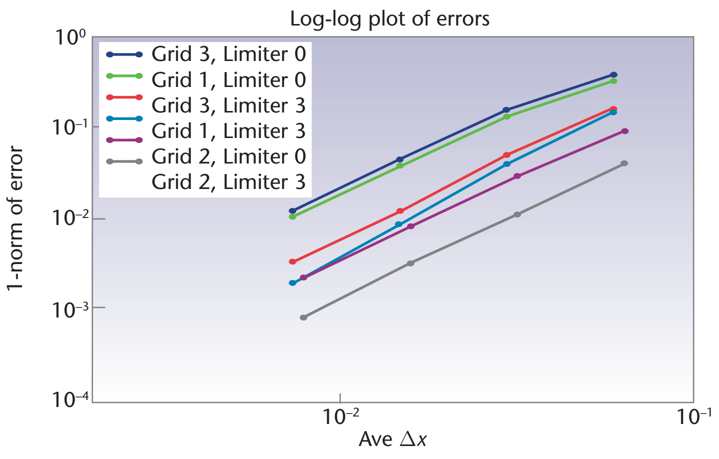

Figure 2. Log-log plot of the errors from 24 test cases. We can run these cases automatically with a single Python script to test the accuracy of the different methods and grid choices by determining the error and observed order of accuracy (presented in Tables 1 through 3) as the grid resolution is varied.

merical methods in each case). We compute the error at time $t = 0.75$ by comparing the numerical solution in each grid cell to the true solution at the cell center. The sum of the absolute value of all errors is scaled by the average cell area to give a value that approximates the 1-norm of the error. The value "Ave Δx" listed in the tables is the square root of the average cell area for the grid. We also present the errors from all 24 tests in a single log-log plot in Figure 2 to more easily compare the accuracy of the methods on different types of grids. A slope of 2 in the log-log plot corresponds to second-order accuracy.

We achieve roughly second-order accuracy on all three grids with either choice of limiter. Errors are smallest in polar coordinates, which isn't surprising because the grid is aligned with the flow, and the advection equation reduces to 1D advection in the angular coordinate θ.

We also see that the error on grid 3 is only slightly larger than on grid 1, the Cartesian grid, verifying that we can indeed use these highly skewed quadrilateral grids with the algorithms implemented in Clawpack. Finally, we observe that, on all three grids, the use of the limiter improves the 1-norm of the error by a factor of three or more.

Python Tools

The Clawpack code is easily set up to run this problem for any one of the test cases described earlier by setting the data parameters appropriately. We could run the 24 tests by hand and collect the results, but with the newly developed Python tools for Clawpack, it's much easier to automate this entire process in a way that can be archived and duplicated at a later time.

Figure 3 shows a simplified version of the Python script to run the numerical tests; it runs 16 tests using only grids of type 1 and 3. The polar grids require a different domain in the computational r and θ coordinates and the specification of periodic boundary conditions in one direction. These modifications are easily accomplished by setting some of the parameters differently; the `clawtest.py` module at www.clawpack.org/links/cise09 provides full details.

The script in Figure 3 uses several functions from the `clawtools` and `clawtest` modules that we don't display here but that appear at the URL listed earlier. The `ClawData` class stores parameter values that are then written into the data files for use in the Fortran code. For example, the file setprob.data contains a line

```
1 =: igrid # which type grid to use
```

and the Python commands

```
data = clawtools.ClawData()
data.igrid = 2
data.write('setprob.data')
```

will replace the value 1 in this data file with 2. All other lines of the data file are unaltered. The only change made to the data files from previous versions of Clawpack is the introduction of the assignment operator `=:` in the portion of each line that the Fortran code ignores (typically, it only reads a single number from each line).

The example I just gave illustrates a modest attempt at providing tools to ease the task of doing reproducible research in the development of numerical methods for hyperbolic problems. Use of these tools requires little overhead or infrastructure beyond the classical use of Clawpack. Python is ideally suited to this purpose and is available on virtually every operating system. The tools could easily be adapted to other contexts, for use with any code that reads parameters from a text file to set up a particular run.

The use of these tools makes it much easier to archive the environment used to generate a series of tests. Rather than requiring the retention of 24 sets of data files, one for each test case, a single Python script clearly shows the parameter choices used for each test case and allows them all to be easily recreated later if needed.

Of course, researchers must also archive the computer code used to perform their tests. If the code changes in the future, having the data sets preserved will be insufficient to reproduce the tests. For ongoing research projects, versioning software such as Subversion (http://subversion.tigris.org) can easily preserve the code's state at any point in time without the need to save full copies of the entire code. By saving the revision number of the code, it's possible to recreate any past state. The use of versioning software should be mandatory among computational scientists at this point. It's easy to use and extremely beneficial, even for a single individual developing programs and writing papers based on the results, and invaluable for any kind of collaborative work.

There's an additional difficulty with attempting to perform reproducible research that I haven't yet addressed—namely, the fact that even if the code and the data are preserved, the computer, operating system, or computer language could change in ways that render the code unusable or unable to reproduce previous results. As an example particularly relevant to the tools described here, the Python language is undergoing revision, and Python 3.X won't be backward compatible with the Python 2.5 used for the tools recently developed. Hence, in addition to archiving the tools, researchers should really archive the current state of the language as well, or at least assume this is being done somewhere (as is the case for Python). Archiving the computer operating system and the computer itself is, of course, more difficult.

Looking further down the road, there's great uncertainty about the durability of digital archives of any sort. Although books and papyrus can last millennia, we often find it's impossible to read data or computer codes from even a few years ago. This is obviously a concern that many professionals are working hard to address, but in the mean time, it's not an excuse to avoid attempting to achieve some level of reproducibility with the available technology.

The Web page I created for this article (www.clawpack.org/links/cise09) also provides a glimpse of the manner in which Python tools can be developed to advance the goals of literate program-

```
import clawtools
from clawtest import *

# special data structure for Clawpack parameters:
data = clawtools.ClawData()
data.tfinal = 0.75 # final time
data.nout = 1 # output solution only at final time

# List parameters for tests to be performed:
grids = [1,3] # set of grids to test
limiters = [0,3] # set of limiters (mthlim values) to test
mxvals = array([30,60,120]) # mx values
myvals = array([30,60,120]) # my values
area = pi # area of circle, for L1 norm

table = {} # dictionary of data and results for each test

for mthlim in limiters:
   data.mthlim = mthlim
   for igrid in grids:
      # Write the value igrid into data file setprob.data:
      data.igrid = igrid;
      data.write('setprob.data')
      # create a dictionary to hold the data and results for this test:
      table[(mthlim,igrid)] = {}

      this_table = table[(mthlim,igrid)] # short name
      this_table['mxvals'] = mxvals # grid resolutions to test
      this_table['myvals'] = myvals
      this_table['ave_cell_area'] = area / (mxvals*myvals)
      this_table['errors'] = empty(len(mxvals)) # filled with results below

      for itest in range(len(mxvals)):
         data.mx = mxvals[itest]; mx = data.mx # short form
         data.my = myvals[itest]; my = data.my # short form
         data.write('claw2ez.data') # write mx,my,tfinal,nout

         # run Fortran code:
         clawtools.runclaw()

         # compute errors:
         errors = compute_errors(frame=data.nout)
         # approx 1-norm of error:
         errorsum = abs(errors).sum()
         error1 = errorsum * this_table['ave_cell_area'][itest]
         this_table['errors'][itest] = error1

# Create Tables 1--3 and Figure 2 of this paper:
make_latex_table(table, limiters, grids, fname='errortables.tex')
make_error_plots(table, limiters, grids, fname='errors.png')
```

Figure 3. The Python script `clawtestsubset.py` for running 16 test problems. The full module `clawtest.py` for all 24 test cases, which also includes other functions, appears at www.clawpack.org/links/cise09.

ming. The hope is to illuminate the algorithms that produced the published results within the computer code's documentation. When documenting mathematical programs, for example, it's convenient to be able to include mathematical descriptions in the code's comments, written in LaTex and readable (as typeset mathematics) along with the code.

We recently developed a general Python script (`mathcode2html.py`) to convert source code in many languages (and data files) into HTML documents, with any comment statements delimited by begin_html and end_html treated as HTML code. When used in conjunction with jsMath (www.math.union.edu/~dpvc/jsMath/), this lets the author include simple LaTex equations in the comments that will be properly typeset in the resulting HTML version of the code. Some wiki-like formatting tools allow links to be included in the computer code in a manner that's fairly readable in the raw code and yet easily converted into the appropriate links in the HTML version. We've included a more specialized version of this script, `clawcode2html.py`, with the latest Clawpack bundle for documenting and cross referencing the code and examples.

Literate programming and reproducible research in computational science often go hand in hand, and people have developed other approaches and systems for implementing some combination of these goals. A few notable projects are the CWEB system (www-cs-faculty.stanford.edu/~knuth/cweb.html), Noweb (www.eecs.harvard.edu/nr/noweb/), Sweave,[8] AMRITA (www.amrita-cfd.org), and Madagascar (www.rsf.sourceforge.net/Main_Page). Nelson Beebe[9] provides a bibliography of papers on literate programming, and a Web search on reproducible research produces many other projects and papers on this topic.

The tools described in this article have one advantage, I believe, over the more ambitious approaches described in some of the works just cited: our tools require relatively little infrastructure and can be added on to existing projects incrementally rather than requiring a fresh start and commitment to a particular large-scale software framework. Although all programs would ideally be designed from the beginning in a literate and reproducible manner, this isn't likely to happen soon, and legacy codes from decades ago will be with us for some time to come. However, appropriate tools can greatly enhance these older codes, as I've attempted to illustrate with Clawpack. In my own work, I plan to use these tools in the future to facilitate writing papers in a more reproducible manner, with the specific version of the code used recorded via a Subversion revision number, the data files and Python scripts used to run the tests archived on a Web page, and new algorithmic features encapsulated in programs with human-readable documentation.

These tools have already gone through several iterations and major rewrites (see www.clawpack.org/links/cise09 for pointers to more recent versions of these tools), but I believe the efforts we're now investing in this will pay off handsomely in the future. With luck, the resulting tools will be easy enough to use on a daily basis that our future research will quite naturally be reproducible with little extra effort. I hope that some of these tools, with minor modifications, will also prove useful for research in other branches of computational science.

Acknowledgments

The introductory part of this article is based on an invited talk and paper that appeared in the Proceedings of the International Conference of Mathematicians *(www.amath.washington.edu/~rjl/pubs/icm06); numerous people provided valuable feedback during the preparation of that paper and in response to it. I also thank the editor and anonymous reviewers, who further helped improve this work. Thanks are due also to Kyle Mandli for assistance in developing the Python and Web portal tools used with Clawpack. This work was supported in part by US National Science Foundation grant DMS-0609661.*

References

1. M. Schwab, N. Karrenbach, and J. Claerbout, "Making Scientific Computations Reproducible," *Computing in Science & Eng.*, vol. 2, no. 6, 2000, pp. 61–67.
2. L.N. Trefethen, *Spectral Methods in Matlab*, SIAM, 2000.
3. "Revised Policy on Enhancing Public Access to Archived Publications Resulting from NIH-Funded Research," Nat'l Institutes of Health, 2008; www.grants.nih.gov/grants/guide/notice-files/NOT-OD-08-033.html.
4. R.J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, 2002.
5. D.I. Ketcheson and R.J. LeVeque, "WENOCLAW: A Higher Order Wave Propagation Method. In Hyperbolic Problems: Theory, Numerics, Applications," *Proc. 11th Intl. Conf. Hyperbolic Problems*, S. Benzoni-Gavage and D. Serre, eds., Springer, 2006, pp. 609–616.
6. D.E. Knuth, "Literate Programming," *The Computer J.*, vol. 27, no. 97, 1984, p. 111.
7. D.A. Calhoun, C. Helzel, and R.J. LeVeque, "Logically Rectangular Finite Volume Grids and Methods for 'Circular' and 'Spherical' Domains," *SIAM Rev.*, vol. 50, 2008, pp. 723–752.
8. F. Leisch, "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis," W. Härdle and B. Rönz, eds., *Compstat 2002—Proc. Computational Statistics*, Physica Verlag, 2002, pp. 575–580.
9. N.H.F. Beebe, "A Bibliography of Literate Programming," 2002; www.literateprogramming.com/litprog-bib.pdf.

Randall J. LeVeque *is a professor of applied mathematics at the University of Washington. His research interests include numerical analysis, nonlinear differential equations, and applications in geo-, bio-, and astrophysics. LeVeque has a PhD in computer science from Stanford University. Contact him at rjl@washington.edu.*