Scientific Tests and Continuous Integration Strategies to Enhance Reproducibility in the Scientific Software Context

Matthew Krafczyk
krafczyk.matthew@gmail.com
University of Illinois at Urbana-Champaign
Champaign, Illinois

August Shi
awshi2@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, Illinois

Adhithya Bhaskar
bhaskar7@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, Illinois

Darko Marinov
marinov@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, Illinois

Victoria Stodden
vcs@stodden.net
University of Illinois at Urbana-Champaign
Champaign, Illinois

ABSTRACT

Continuous integration (CI) is a well-established technique in commercial and open-source software projects, although not routinely used in scientific publishing. In the scientific software context, CI can serve two functions to increase reproducibility of scientific results: providing an established platform for testing the reproducibility of these results, and demonstrating to other scientists how the code and data generate the published results. We explore scientific software testing and CI strategies using two articles published in the areas of applied mathematics and computational physics. We discuss lessons learned from reproducing these articles as well as examine and discuss existing tests. We introduce the notion of a *scientific test* as one that produces computational results from a published article. We then consider full result reproduction within a CI environment. If authors find their work too time or resource intensive to easily adapt to a CI context, we recommend the inclusion of results from reduced versions of their work (e.g., run at lower resolution, with shorter time scales, with smaller data sets) alongside their primary results within their article. While these smaller versions may be less interesting scientifically, they can serve to verify that published code and data are working properly. We demonstrate such reduction tests on the two articles studied.

CCS CONCEPTS

• **General and reference** → **Validation**; **Verification**; • **Software and its engineering** → *Software reliability*; *Software usability*.

KEYWORDS

Reproducibility; Continuous Integration; Software Testing; Software Reliability; Scientific Software

P-RECS'19, June 24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6756-1/19/06.
https://doi.org/10.1145/3322790.3330595

ACM Reference Format:
Matthew Krafczyk, August Shi, Adhithya Bhaskar, Darko Marinov, and Victoria Stodden. 2019. *Scientific Tests* and Continuous Integration Strategies to Enhance Reproducibility in the Scientific Software Context. In *2nd International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS'19), June 24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3322790.3330595

1 INTRODUCTION

Reproducibility is baked into the scientific method and modern science. In a healthy research community it is expected that a certain fraction of published results will fail to hold up as knowledge evolves, however the scientific community is questioning the current rate of irreproducibility [22, 24] and pushing to improve the reproducibility of published works [4, 9, 14, 40]. Recent studies, however, have shown reproduction failures even with new policies meant to improve them [16, 43]. These failures highlight the need for new standards and practices to ensure transparency and computational reproducibility of data- and computationally-enabled results [41]. This article is intended to contribute to the search for new techniques to mitigate these failures.

We first clarify what we mean by "reproducibility" in this work [4]. Here we refer to whether another scientific team can produce the same computed values as the original team using the same methods and input data, in other words computational reproducibility [37]. In the computational context, a different researcher should produce identical or nearly identical (within reasonable bounds for non-deterministic computations [7]) values from the same code and input data. A published article is computationally reproducible if the same values can be produced using the original code and data. Importantly, we are not interested in arbitrating the scientific accuracy of specific code or algorithms.

Reliable software testing forms a cornerstone of modern software engineering. Good tests allow the engineer to detect problems that may be introduced while code is written and changed. Software testing comes in several flavors, to name a few: white-box or black-box testing and unit or system testing [30, 34]. Over time, the nature of testing has changed, but general purpose techniques such as continuous integration (CI), are now central to the software testing process [6, 20, 21].

Most scientific code must undergo a process of verification where its functioning is compared to analytic solutions of theoretical models the code approximates [34]. Scientists usually publish their verification process within their article, linking scientific code verification closely to reproducibility. An example is found in computational physics: a numerical model created for a differential equation describing an idealized system. This numerical model or "scheme," is then translated into code by the scientist. The code is verified by checking its result against an analytic solution whenever possible [34]. This check examines the functionality of the scheme and its implementation, and whether it matches its intended use.

White-box and unit testing are ill-suited for this type of verification as they determine whether specific functions or sections of code are functioning properly, rather than the entire code. Black-box or system testing however, seeks to test the whole application [30]. Although not routinely used in scientific research, the verification process is very similar and often reproduces results of an article. Tools and techniques that make black-box testing easier to use, and access therefore improve reproducibility.

Container environments like Docker [32] provide a uniform virtualized environment. Many CI environments such as Travis CI [1] use containerization as part of their backend. CI and containers are ideal environments to test reproducibility of scientific works. A uniform environment frees the author from worrying about the environment of the user. Starting empty, the author must specify the necessary computational environment as well as how their tests should be run.

From a previous study on computational reproducibility [39], we encountered two articles suitable for showcasing testing in the context of a scientific work. After reproducing their results, we worked to implement the reproduction as a CI test. With author permission, we release software packages for each article.

There are many efforts studying and influencing the state of reproducibility. Various tools exist that can help alleviate reproducibility problems encountered by scientists [8, 10, 11, 13, 26, 28, 29, 31–33]. Scientists and committees create recommendations for changes in practice that can improve the reproducibility of final results [2, 3, 5, 12, 15, 18, 23, 27, 35, 36, 38, 42, 44]. We consider this article to extend recommendation efforts, clarifying how software testing and CI can be made useful for scientists.

Many scientists may encounter high resource usage or runtimes as a barrier to reproduction within a CI environment. In these cases, we recommend authors augment their primary results with those of reduced versions of their work. For example, running a simulation at reduced resolution will reduce memory usage as well as run faster. Such minimized tests serve to verify that published code and data are working properly. Another strategy is to provide shorter running tests to get immediate feedback alongside longer running tests that may reveal more difficult to detect bugs with more run time.

In this article we first discuss our methods for reproducing the original articles in Section 2. Several ideas are explored in the Discussion section (Section 3): problems encountered during reproduction, what makes a good scientific test, benefits and strategies for implementing tests with differing lengths, issues related to the community, future work, and our code and data release information. We finish with some concluding remarks in Section 4.

2 METHODS

During previous investigations, we considered the reproducibility of 306 computational physics articles published in the Journal of Computational Physics, and collected code and data from 55 of those articles [39]. In ongoing work, we considered how much of each article could be reproduced with up to 40 hours of dedicated effort. For this study, we selected two articles for which we were able to reproduce most of the article, and the authors gave us permission to share their code (essential to demonstrate our CI strategies).

We describe the subject matter of each article. The first article [45] we study provides a new method for inverting the factored eikonal equation. The eikonal equation describes the propagation of waves through a medium, so inverting it can be used to measure properties of the medium [45]. The second article [19] concerns a modification of Newton's method for non-differentiable functions.

Article [45] demonstrated good scientific testing by implementing broad tests like the code verification tests mentioned above. The authors also use Travis CI [1], a popular CI service for open-source projects.[1] We implemented our own tests for article [19], as no code or data was published. For each article, we enumerated a list of figures and tables which carried results from computational experiments, and used existing code or wrote code to complete the reproduction within 40 hours.

When inspecting the released code for article [45] we found a script `runExperiments.jl` that reproduced Tables 1-3 from the article. However, the script crashed during the higher resolution experiments. Since their code was released under the MIT license, we forked their repository and made changes to reproduce more figures and skip experiments that took too long or required excessive resources. All table values were reproduced along with Figure 3. Table 1 shows our reproduction of Table 3 from article [45]. Other figures from the article were missing critical data.

As mentioned, article [19] was published without code we implemented their solver in Octave and created tests matching those published in their article. We were able to reproduce their results from all tables to machine precision.

With both articles of interest successfully reproduced, we focused on constructing testing scripts capable of running the whole procedure on our local machines, within a Docker [32] container, and within the Travis CI environment. We define these tests as *scientific tests* – tests that produce computational results from a published article. Scientific tests by definition follow the mold of the black-box test, and the most expansive scientific test is to simply reproduce all of the results from the published article. We elaborate more on this testing philosophy in Section 3.

Within each script, we first run the computational experiment or experiments, then compare the results against a known good solution: the results in the published article. Choosing which computational experiments to run and how to compare their results to the known solution is critical. While we built our test scripts to allow for full reproduction, we enabled the user to truncate the tests if sufficient time or resources are unavailable. On Travis CI, we run only those experiments which can complete within 5 minutes, well under Travis's 20 minute polling interval. Once complete, the

[1] Their `.travis.yml` file, the configuration file for Travis CI, is at https://github.com/JuliaInv/FactoredEikonalFastMarching.jl/blob/master/.travis.yml

Table 1: Our reproduction of Table 3 from article [45]. We draw the reader's attention to the increasing computational cost (columns listed as time) towards the bottom of the Table. Upper rows are suitable for quickly completing tests, while lower rows should be allowed more time.

h	n	1^{st} order error in τ	time (work)	2^{nd} order error in τ	time (work)
1/40	161×321	[6.15e-03,3.86e-03]	0.06s (309.15)	[1.60e-04,5.94e-05]	0.06s (309.86)
1/80	321×641	[1.54e-03,9.67e-04]	0.25s (329.09)	[3.85e-05,1.56e-05]	0.25s (298.17)
1/160	641×1281	[1.54e-03,9.67e-04]	1.02s (329.09)	[1.08e-05,4.03e-06]	1.09s (351.82)
1/320	1281×2561	[7.68e-04,4.83e-04]	4.45s (362.82)	[3.18e-06,1.04e-06]	4.58s (373.24)
1/640	2561×5121	[3.84e-04,2.42e-04]	18.53s (371.54)	[9.59e-07,2.66e-07]	19.06s (382.16)
1/1280	5121×10241	[1.92e-04,1.21e-04]	77.53s (387.00)	[2.99e-07,6.88e-08]	80.79s (403.26)

results must be compared with the article. For this, we implemented a Python script to extract results from the output and compare them to the expected values within a small tolerance for error.

Our test scripts follow a simple hierarchical structure. First, a master script named `run.sh` coordinates all aspects of the experiment from start to finish. This `run.sh` calls subordinate scripts to perform different stages of the work such as running the computational experiment proper and the comparison of the results to known good results. This structure makes the experimental procedure clear, easing future modifications[2]. Once the testing scripts were complete, we produced a `.travis.yml`[3] file. It uses the Ubuntu OS's repositories to install all necessary software, and then execute the master `run.sh` script. Computational experiments are run, their results are checked, and Travis CI reports success or failure status.

3 DISCUSSION

Code from [19] uses Travis CI and we extend this work to conform with the notion of *scientific tests* we introduce in this article. We discuss limitations of these tests in Section 3.1.The challenges we encountered reproducing the results for use in Travis CI are discussed in Section 3.2. In Section 3.3, we elaborate on the benefits of scientific testing and why scientists should implement reproductions of their published results as the test of choice. Strategies for dealing with long-running or resource-consuming experiments in CI environments are discussed in Section 3.4. How our work interacts with current practices and industry efforts is discussed in Section 3.5. We mention future avenues of investigation Section 3.6 and we discuss our code and data release in Section 3.7.

3.1 Existing tests and their limitations

The authors of article [45] generously provide their code through a GitHub repository[4] with a free software license. Inspection of their code shows that the authors have already created a Travis CI configuration file called `.travis.yml`. This file configures Travis CI to download and install version 0.6 of Julia and then executes a test suite that is built into their generalized library solving factored Eikonal equations.

Our inspection of the authors' test suite revealed that they perform computational experiments that are *similar but not the same* as those published in the article [45]. Their tests perform versions of the experiment called 'Test case 1' in their article. The test version of the experiment has a significantly lower resolution, and the results are compared against an analytical solution similar to Tables 1-6 from their article.

This set of tests follows the spirit of black-box scientific testing we introduce, but could go farther. First, the experiments being run differ from those in the published article in their resolution and so results from these experiments do not appear in the original article. For outside observers using this code for the first time, it can be difficult to determine whether passing this set of tests is enough to know whether the code will faithfully reproduce what was published. Had these parameters matched even just the top row of Table 1 from [45], readers could be much more confident that their numbers would match those in the article.

We modified their `.travis.yml` to execute our dedicated `run.sh` script that runs the computational experiments from the paper. The script allows an option for the number of rows to reproduce, so Travis only executes some of the table rows. Finally, we added logic that checks whether the output of the script matches the numbers from the article, within a 10% tolerance.

3.2 Reproduction Challenges

Reproducing the two articles [19, 45] both came with their challenges. For [19], we originally believed the computations in the article were simple enough that changes to Octave would present exactly the same answer. This allowed us to apply a simple diff between the computed and expected answers. Unfortunately, this was not the case. First, the variables R and d from the article had slight differences near the machine precision level. Table 2 shows these differences at around 10^{-8}. However, our test for differences must now accommodate this. We produced a Python script to compare the computed values to the published values and check they are equivalent within an error tolerance.

While reproducing [45], we encountered two notable problems. First, the code published relied on an older version of Julia, version 0.6. The easiest and most reliable method to get this version of Julia was to use the tarball download of the pre-compiled binaries.

[2]The `run.sh` file we created for article [19], can be found here: https://raw.githubusercontent.com/ReproducibilityInPublishing/10.1016_S0377-0427-03-00650-2/master/run.sh

[3]You can find the `.travis.yml` for article [19] here: https://raw.githubusercontent.com/ReproducibilityInPublishing/10.1016_S0377-0427-03-00650-2/master/.travis.yml

[4]https://github.com/JuliaInv/FactoredEikonalFastMarching.jl, commit `ef03de` at the time our reproduction was performed.

Table 2: Values of R and d from article [19] varied depending on which Octave version was used.

OS	Octave Version	R	d
Ubuntu 14.04	3.8.1	0.00837735	0.41411902
Ubuntu 16.04	4.0.0	0.00837733	0.41411889
Arch	5.1.0	0.00837733	0.41411889

Building it ourselves or obtaining older versions through repositories was unsuccessful or very difficult. Second, we did not initially find code to reproduce plots from the article such as Figure 3. By examining the git history of the project, we were able to recover older code which produced similar plots. We adapted this code to produce figures exactly like those published. Finally, although we reproduced Figure 3, we could not perform automatic comparison to the published version. The authors did not publish the data behind their figure, and we could not properly compare images as slight differences to rendered colors could affect the results. Such a detailed procedure was beyond the scope of this work.

3.3 Black-box Scientific Testing

Black-box testing is conceptually a test that checks the behavior of software rather than deriving tests based on the code itself. Tests in this paradigm do not focus on specific aspects of individual functions or methods in the software but rather test broad behavior of the software without knowing how it was implemented. When publishing scientific articles, authors could include code verification tests appropriate to their scientific domain to justify the correctness of their code and advocate the use of their method in the wider community. Usually, these code verification tests are fairly general, and make good candidates for black-box tests. To help illustrate, we discuss such tests in the context of our two articles of interest.

The authors of [19] first describe their method theoretically, establishing the method's behavior and convergence, then present several different example inputs to pass through the method. These inputs consist of two different systems of equations: Examples 1 and 2 from their article. Each example shows how the method converges to an answer and how the performance of the method may be optimized by tweaking method parameters. The presentation culminates in Tables 1–3 from the article that show how the method behaves at each iteration with a range of method parameters. Both examples offer a comprehensive test of the methods behavior, and should an implementation follow the behavior listed, one can be confident it reflects what the authors described in their article. Thus reproducing the tables from their article forms a rigorous black-box scientific test.

The authors of article [45] demonstrate their method with a series of examples using analytically defined mediums. Travel times are extracted between pairs of randomized points and the method is applied to extract the medium. The results of these experiments are summarized in Tables 1–6 in their article. The comparison of extracted to analytic travel times constitutes scientific code verification, and the agnostic nature of the test to implementation details constitutes a black-box scientific test. Similarly to the previous article, reproducing the results published in these tables makes a good black-box scientific test.

Unfortunately, several factors prevent the bit-wise comparison of published results and computed results. By "bit-wise comparison" we mean performing a simple difference test with a failure occurring whenever the two numbers are different at all. First, code can evolve with time, introducing changes into necessary tools or dependencies that can change the results. In particular, various operations on floating-point numbers introduced imprecision in these different runs. Second, certain multi-threaded algorithms are inherently non-deterministic, which can result in small changes between calculated results and those published, although we note that this was not the case for us here. As mentioned in Section 3.2, we encountered the first type of change for the second article [19]. Attempting to control all these sources of non-determinism is very difficult, so the most pragmatic way to handle them would be to anticipate them.

Anticipating acceptable changes to scientific results is challenging and requires exact knowledge of the algorithm itself. The original author is usually in the best position to provide an acceptable bound. In the case of non-determinism, re-running the analysis multiple times can also provide an estimate of resultant changes. Because we were not the original authors, and our articles did not involve non-determinism, we did not provide an accurate tolerance. Instead, we chose a loose bound of 10%, which allows for small changes, such as from the changes in the underlying libraries, but fails for significant changes.

3.4 Minimized Testing

Some articles may not include results computable with consumer hardware, requiring both an extreme amount of time and resources. We recommend these authors publish results from 'minimized' versions of their computational experiments together with their main results. With minimized computational experiments, serious science changing bugs can be detected quickly. Fast access to such information is worth the extra effort of creating the minimized test.

The most successful strategy for designing a minimized test is to mimic the original work as much as possible but at a smaller scale. If the code takes too long to run, then scaling down the resolution and/or reducing the amount of time steps simulated can both result in faster running simulations. If the code writes too much data to disk for example, the author can aim to reduce how often snapshots are saved or reduce resolution or time steps. Such tests provide extensive groundwork for readers to understand and execute the full set of simulations, and are subject to the same types of variability as the full simulation.

To make this scheme more clear, consider our first article [45]. The authors show the results of their proposed method with a series of tables, e.g., their Table 3 reproduced here as Table 1. Each row in the table shows the error between the analytical result and the computed result for different sets of parameters, with the computational cost of each experiment increasing towards the bottom of the table, as shown by increasing computing times. The first rows are small, short experiments that can be completed quickly even on a system with low resources. The rows at the bottom start to take several minutes and require a lot of memory to finish. Had the authors not included the earlier rows, which are easier to complete, we would have suggested that they purposely lower the resolution of their simulation and include those results as well.

Travis CI offers the `stages` mechanism to define different test categories. Tests for each of the defined stages are executed in sequence, and certain stages can be flagged to not trigger a build failure when they do fail. This technique can be harnessed to create two test groups. A first group completes quickly, letting the authors know about obvious bugs. These are often called "smoke tests" or "build verification tests", and in our paradigm, minimized scientific tests are a good fit. A second group of tests can take much longer to complete performing more thorough analysis. Scientific tests that require a large amount of time, but for which authors do not want to wait for their completion during the software development cycle, are good candidates here.

The code for [19] runs too quickly to make a meaningful distinction, but [45] has results that can take almost 5 minutes to run. We have grouped reproduction of all but the last two rows of Tables 1–6 from [45] as the smoke tests in this paradigm, and the last two rows of each table as the set of long-running tests.

3.5 Prior Work and Community Recommendations

Here, we mention some especially relevant prior work in the community as well as give some specific community guidelines. We want to highlight Popper [26] and their related tool Popper CI [25]. The authors of Popper propose the creation of dedicated scripts to check produced solutions appropriate for their specific situation. We note that this is very similar to what we are proposing here. As many researchers use Jupyter notebooks, integration of those workflows into CI systems should be addressed. We recommend converting the notebook to a script using `nbconvert` and then executing this script and checking its output. We note that this is not a fool-proof method as Jupyter notebook specific 'magics' may not work properly as a simple script. Existing CI systems could be improved for scientific software reproduction by the addition of non-binary failure states. For example, rather than a test simply failing or succeeding, the CI system could return a partial success perhaps with a percentage.

3.6 Future Work

We have left several avenues of investigation open for further work. First, CI environments utilize hardware resources which may change transparently to the user. We did not measure the effect of such a change as we cannot control when Travis CI performs them, and the additional scripting and data storage infrastructure necessary for such a study was prohibitively time-consuming to implement. Data of this type however could reveal whether updated hardware can change results as well. Second, additional testing methods such as unit and white-box testing could be tested more carefully to gain a more complete picture of the testing landscape. Finally, this study only concerns two articles. An expansion not only of the number of articles considered, but also their subject matter could reveal more general testing techniques or highlight special cases for certain workflows.

3.7 Code and Data Release

All code and data relied on for this article are available as public repositories on GitHub. Both packages provide a `Dockerfile` with which an appropriate Docker container can be built, and a `run.sh` script that can be executed to perform all or most computational experiments from the article. All experiments can be performed without using a Docker container, but all necessary software will need to be installed. A list of software necessary can be found in the `Dockerfile` instructions. Each article of interest's code is also checked into GitHub. All of these packages also include a `.travis.yml` file as to define how to build and run the tests on Travis CI. Relevant links for each article are found below:

- Article [45]
  - Code and data: https://github.com/ReproducibilityInPublishing/j.jcp.2016.08.012. (Commit `ba16911` at time of publication)
  - Travis CI: https://travis-ci.org/ReproducibilityInPublishing/j.jcp.2016.08.012
- Article [19]
  - Code and data: https://github.com/ReproducibilityInPublishing/10.1016_S0377-0427-03-00650-2. (Commit `227b842` at time of publication)
  - Travis CI: https://travis-ci.org/ReproducibilityInPublishing/10.1016_S0377-0427-03-00650-2

4 CONCLUSIONS

Data- and computationally-enabled science can be made more robust by adapting software engineering and software testing techniques. Continuous Integration (CI) can provide important benefits to the scientific reproduction process including for example a uniform platform on which to build instructions and execute a reproduction. We examined two articles [19, 45] to understand how traditional software testing can advance reproducibility in a scientific context. We introduce black-box scientific testing, where scientific results from a computation are compared to those published to provide the most reliable indication that a scientific work is being faithfully represented by the code.

When scientists produce code for experiments that are too time or resource intensive to run in a CI environment, we proposed the production of minimized computational experiments that they can publish in the same article as their main results. The cost in time spent to implement these additional tests is worth the effort because they can expose not only whether the software is working properly even when the authors are making changes, but also whether the results are sensitive to the computation environment. As more scientists adopt these changes into their workflows we can expect to see publications that are more easily verified and as a by-product more reliable, strengthening the scientific record.

ACKNOWLEDGMENTS

Funding for this research was provided by NSF Awards CCF-1421503, CCF-1763788, and OAC-1839010. We thank NCSA and their SPIN program for their support. We also thank the anonymous reviewers who helped us to improve this manuscript.

REFERENCES

[1] [n.d.]. Travis CI. https://travis-ci.org

[2] David Bailey, Jonathan Borwein, and Victoria Stodden. 2013. Set the default to 'open. *Notices of the AMS, Accepted March* (2013). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.310.4101

[3] Lorena A Barba. 2018. Praxis of Reproducible Computational Science. https://doi.org/10.22541/au.153922477.77361922

[4] Lorena A. Barba. 2018. Terminologies for Reproducible Research. *ArXiv e-prints* (Feb 2018). arXiv:cs.DL/1802.03311

[5] Francine Berman, Rob Rutenbar, Brent Hailpern, Henrik Christensen, Susan Davidson, Deborah Estrin, Michael Franklin, Margaret Martonosi, Padma Raghavan, Victoria Stodden, and Alexander S. Szalay. 2018. Realizing the Potential of Data Science. *Commun. ACM* 61, 4 (March 2018), 67–72. https://doi.org/10.1145/3188721

[6] G. Booch. 1991. *Object Oriented Design: With Applications*. Benjamin/Cummings Pub. https://books.google.com/books?id=w5VQAAAAMAAJ

[7] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, Dong H. Ahn, and Gregory L. Lee. 2013. Determinism and Reproducibility in Large-Scale HPC Systems. https://www.semanticscholar.org/paper/Determinism-and-Reproducibility-in-Large-Scale-HPC-Chiang-Gopalakrishnan/9e8ea7d54dc67f672b31b223ed14edc758b0b28d

[8] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2085–2088. https://doi.org/10.1145/2882903.2899401

[9] Jon F. Claerbout and Martin Karrenbach. 1992. Electronic documents give reproducible research a new meaning. In *SEG Technical Program Expanded Abstracts*. 601–604. https://doi.org/10.1190/1.1822162

[10] K. Cranmer and L. Heinrich. 2017. Yadage and Packtivity - analysis preservation using parametrized workflows. *ArXiv e-prints* (June 2017). arXiv:physics.data-an/1706.01878

[11] K. Cranmer and I. Yavin. 2011. RECAST — extending the impact of existing analyses. *Journal of High Energy Physics* 4, Article 38 (April 2011), 38 pages. https://doi.org/10.1007/JHEP04(2011)038 arXiv:hep-ex/1010.2506

[12] T. Crick, B.A. Hall, and S. Ishtiaq. 2017. Reproducibility in Research: Systems, Infrastructure, Culture. *Journal of Open Research Software* 5 (2017), 32. Issue 1. https://doi.org/10.5334/jors.73

[13] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature Biotechnology* 35 (11 Apr 2017), 316 – 319. http://dx.doi.org/10.1038/nbt.3820

[14] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden. 2009. Reproducible Research in Computational Harmonic Analysis. *Computing in Science Engineering* 11, 1 (Jan 2009), 8–18. https://doi.org/10.1109/MCSE.2009.15

[15] R.R. Downs, W.C. Lenhardt, E. Robinson, E. Davis, and N. Weber. 2015. Community Recommendations for Sustainable Scientific Software. *Journal of Open Research Software* 3 (2015), e11. Issue 1. https://doi.org/10.5334/jors.bt

[16] Bryan T. Drew, Romina Gazis, Patricia Cabezas, Kristen S. Swithers, Jiabin Deng, Roseana Rodriguez, Laura A. Katz, Keith A. Crandall, David S. Hibbett, and Douglas E. Soltis. 2013. Lost Branches on the Tree of Life. *PLOS Biology* 11, 9 (09 2013), 1–5. https://doi.org/10.1371/journal.pbio.1001636

[17] Yuezheng Gong, Qi Wang, Yushun Wang, and Jiaxiang Cai. 2017. A conservative Fourier pseudo-spectral method for the nonlinear Schrödinger equation. *J. Comput. Phys.* 328 (2017), 354 – 370. https://doi.org/10.1016/j.jcp.2016.10.022

[18] Seth Green. [n.d.]. Five reproducibility lessons from a year of reviewing compute capsules. https://medium.com/codeocean/five-reproducibility-lessons-from-a-year-of-reviewing-compute-capsules-de71729ebd8a

[19] M.A Hernández and M.J Rubio. 2004. A modification of Newton's method for nondifferentiable equations. *J. Comput. Appl. Math.* 164-165 (2004), 409 – 417. https://doi.org/10.1016/S0377-0427(03)00650-2 Proceedings of the 10th International Congress on Computational and Applied Mathematics.

[20] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[21] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[22] John P. A. Ioannidis. 2005. Why Most Published Research Findings Are False. *PLOS Medicine* 2, 8 (08 2005). https://doi.org/10.1371/journal.pmed.0020124

[23] Damien Irving. [n.d.]. Best practices for scientific software. https://software.ac.uk/blog/2017-11-29-best-practices-scientific-software

[24] Ioannidis JA. 2005. Contradicted and initially stronger effects in highly cited clinical research. *JAMA* 294, 2 (2005), 218–228. https://doi.org/10.1001/jama.294.2.218 arXiv:/data/journals/jama/4983/joc50060.pdf

[25] I. Jimenez, A. Arpaci-Dusseau, R. Arpaci-Dusseau, J. Lofstead, C. Maltzahn, K. Mohror, and R. Ricci. 2017. PopperCI: Automated reproducibility validation. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 450–455. https://doi.org/10.1109/INFCOMW.2017.8116418

[26] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. [n.d.]. Standing on the Shoulders of Giants by Managing Scientific Experiments Like Software. *;login: The USENIX Magazine* 41, 4 ([n. d.]). https://www.usenix.org/publications/login/winter2016/jimenez

[27] D.S. Katz, K.E. Niemeyer, S. Gesing, L. Hwang, W. Bangerth, S. Hettrick, R. Idaszak, J. Salac, N. Chue Hong, S. Núñez-Corrales, A. Allen, R.S. Geiger, J. Miller, E. Chen, A. Dubey, and P. Lago. 2018. Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4). *Journal of Open Research Software* 6 (2018), 10. Issue 1. https://doi.org/10.5334/jors.184

[28] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. *Jupyter Notebooks – a publishing format for reproducible computational workflows*. 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[29] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20. https://doi.org/10.1371/journal.pone.0177459

[30] Milind G Limaye. 2009. *Software testing*. Tata McGraw-Hill Education.

[31] B. Ludaescher, K. Chard, N. Gaffney, M. B. Jones, J. Nabrzyski, V. Stodden, and M. Turk. 2016. Capturing the "Whole Tale" of Computational Research: Reproducibility in Computing Environments. *Arxiv: CoRR* (2016). https://arxiv.org/abs/1610.09958

[32] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[33] H. Monajemi, D. L. Donoho, and V. Stodden. 2016. Making massive computational experiments painless. In *2016 IEEE International Conference on Big Data (Big Data)*. 2368–2373. https://doi.org/10.1109/BigData.2016.7840870

[34] William L Oberkampf and Christopher J Roy. 2010. *Verification and validation in scientific computing*. Cambridge University Press.

[35] National Academies of Sciences Engineering and Medicine. 2019. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC. https://doi.org/10.17226/25303

[36] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9, 10 (10 2013), 1–4. https://doi.org/10.1371/journal.pcbi.1003285

[37] Victoria Stodden. 2013. Resolving Irreproducibility in Empirical and Computational Research. IMS Bulletin. http://bulletin.imstat.org/2013/11/resolving-irreproducibility-in-empirical-and-computational-research/.

[38] V. Stodden, D.H. Bailey, J. Borwein, R.J. LeVeque, W. Rider, and W. Stein. 2013. Setting the default to reproducible: Reproducibility in computational and experimental mathematics. http://www.davidhbailey.com/dhbpapers/icerm-report.pdf

[39] Victoria Stodden, Matthew S. Krafczyk, and Adhithya Bhaskar. 2018. Enabling the Verification of Computational Results: An Empirical Evaluation of Computational Reproducibility. In *Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS'18)*. ACM, New York, NY, USA, Article 3, 5 pages. https://doi.org/10.1145/3214239.3214242

[40] Victoria Stodden, Friedrich Leisch, and Roger D. Peng. 2014. *Implementing Reproducible Research*. CRC Press.

[41] Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P.A. Ioannidis, and Michela Taufer. 2016. Enhancing reproducibility for computational methods. *Science* 354, 6317 (2016), 1240–1241. https://doi.org/10.1126/science.aah6168 arXiv:http://science.sciencemag.org/content/354/6317/1240.full.pdf

[42] Victoria Stodden and Sheila Miguez. 2013. Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research. *JORS* (Sep 2013). https://doi.org/10.2139/ssrn.2322276

[43] Victoria Stodden, Jennifer Seiler, and Zhaokun Ma. 2018. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences* 115, 11 (2018), 2584–2589. https://doi.org/10.1073/pnas.1708290115 arXiv:http://www.pnas.org/content/115/11/2584.full.pdf

[44] Morgan Taschuk and Greg Wilson. 2017. Ten simple rules for making research software more robust. *PLOS Computational Biology* 13, 4 (04 2017), 1–10. https://doi.org/10.1371/journal.pcbi.1005412

[45] Eran Treister and Eldad Haber. 2016. A fast marching algorithm for the factored eikonal equation. *J. Comput. Phys.* 324 (2016), 210 – 225. https://doi.org/10.1016/j.jcp.2016.08.012