www.software.ac.uk

Managing Research Software Development *– better software, better research*

Slides: https://doi.org/10.6084/m9.figshare.5930662

1st March 2018, Leibniz MMS Days 2018, Leipzig

Neil Chue Hong (@npch), Software Sustainability Institute
ORCID: 0000-0002-8876-7606 | N.ChueHong@software.ac.uk

Supported by BBSRC EPSRC Pioneering research E·S·R·C Economic & Social Research Council

Project funding from Jisc NERC Science of the Environment

Can you spot the mistake?

www.software.ac.uk

	B	C	I	J	K	L	M
2			Real GDP growth				
3			Debt/GDP				
4	Country	Coverage	30 or less	30 to 60	60 to 90	90 or above	30 or less
26			3.7	3.0	3.5	1.7	5.5
27	Minimum		1.6	0.3	1.3	-1.8	0.8
28	Maximum		5.4	4.9	10.2	3.6	13.3
29							
30	US	1946-2009	n.a.	3.4	3.3	-2.0	n.a.
31	UK	1946-2009	n.a.	2.4	2.5	2.4	n.a.
32	Sweden	1946-2009	3.6	2.9	2.7	n.a.	6.3
33	Spain	1946-2009	1.5	3.4	4.2	n.a.	9.9
34	Portugal	1952-2009	4.8	2.5	0.3	n.a.	7.9
35	New Zealand	1948-2009	2.5	2.9	3.9	-7.9	2.6
36	Netherlands	1956-2009	4.1	2.7	1.1	n.a.	6.4
37	Norway	1947-2009	3.4	5.1	n.a.	n.a.	5.4
38	Japan	1946-2009	7.0	4.0	1.0	0.7	7.0
39	Italy	1951-2009	5.4	2.1	1.8	1.0	5.6
40	Ireland	1948-2009	4.4	4.5	4.0	2.4	2.9
41	Greece	1970-2009	4.0	0.3	2.7	2.9	13.3
42	Germany	1946-2009	3.9	0.9	n.a.	n.a.	3.2
43	France	1949-2009	4.9	2.7	3.0	n.a.	5.2
44	Finland	1946-2009	3.8	2.4	5.5	n.a.	7.0
45	Denmark	1950-2009	3.5	1.7	2.4	n.a.	5.6
46	Canada	1951-2009	1.9	3.6	4.1	n.a.	2.2
47	Belgium	1947-2009	n.a.	4.2	3.1	2.6	n.a.
48	Austria	1948-2009	5.2	3.3	-3.8	n.a.	5.7
49	Australia	1951-2009	3.2	4.9	4.0	n.a.	5.9
50							
51			4.1	2.8	2.8	=AVERAGE(L30:L44)	

"All I can hope is that future historians note that one of the core empirical points providing the intellectual foundation for the global move to austerity in the early 2010s was based on someone accidentally not updating a row formula in Excel" – Mike Konczal

- Reinhart, Carmen M.; Rogoff, Kenneth S. (2010). "Growth in a Time of Debt". American Economic Review. 100 (2): 573–78. doi:10.1257/aer.100.2.573
- https://qz.com/75035/fixing-this-excel-error-transforms-high-debt-countries-from-recession-to-growth/
- http://www.nytimes.com/2013/04/26/opinion/debt-growth-and-the-austerity-debate.html
- https://www.bloomberg.com/news/articles/2013-04-18/faq-reinhart-rogoff-and-the-excel-error-that-changed-history

SCIENTIFIC PUBLISHING

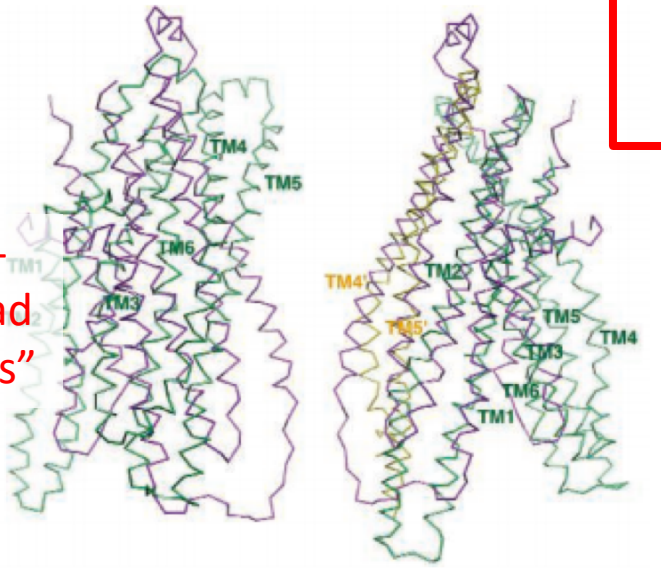
A Scientist's Nightmare: Software Problem Leads to Five Retractions

Until recently, Geoffrey Chang's career was on a trajectory most young scientists only dream about. In 1999, at the age of 28, the protein crystallographer landed a faculty position at the prestigious Scripps Research Institute in San Diego, California. The next year, in a ceremony at the White House, Chang received a Presidential Early Career Award for Scientists and Engineers, the country's highest honor for young researchers. His lab generated a stream of high-profile papers detailing the molecular structures of important proteins embedded in cell membranes.

Then the dream turned into a nightmare. In September, Swiss researchers published a paper in [illegible] protein structure Chang's group had published in a 2001 *Science* paper. When he investigated

Chang was horrified to discover that a homemade data-analysis program had flipped two columns of data, inverting the electron-density map from which his team had derived the final protein structure. Unfortunately, his group had used the program to analyze data for

2001 *Science* paper, which described the structure of a protein called MsbA, isolated from the bacterium *Escherichia coli*. MsbA belongs to a huge and ancient family of molecules that use energy from adenosine triphosphate to transport molecules across cell membranes. These so-called ABC transporters perform many



Flipping fiasco. The structures of MsbA (purple) and Sav1866 (green) overlap little (*left*) until MsbA is inverted (*right*).

Sciences and a 2005 *Science* paper, described EmrE, a different type of transporter protein.

Crystallizing and obtaining structures of five membrane proteins in just over 5 years was an incredible feat, says Chang's former postdoc adviser Douglas Rees of the California Institute of Technology in Pasadena. Such [illegible] because they are large, unwieldy, and notoriously difficult to crystallize. [illegible] needed for x-ray crystallography. Rees says [illegible] Chang's success. "He has an incredible drive and work ethic. He really pushed the field in the sense of getting things to crystallize that no one else had been able to do." Chang's data are good, Rees says, but the faulty software threw everything off.

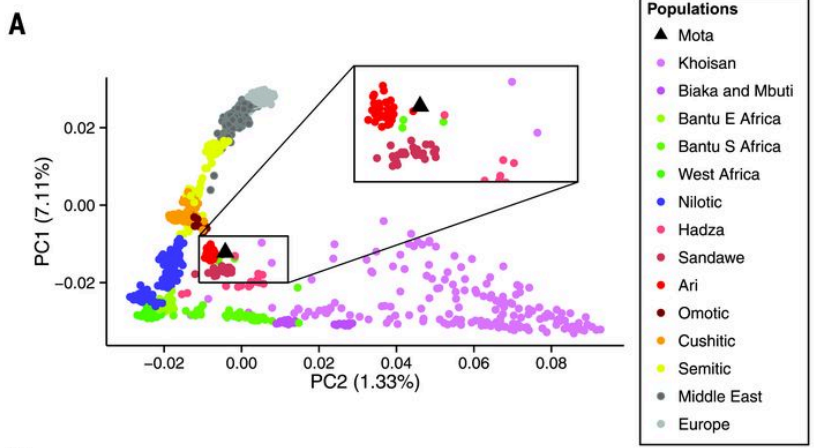
Ironically, another former postdoc in Rees's lab, Kaspar Locher, exposed the mistake. In the 14 September issue of *Nature*, Locher, now at the Swiss Federal Institute of Technology in Zurich, described the structure of an ABC transporter called Sav1866 from *Staphylococcus aureus*. The structure was dramatically—and unexpectedly—different from that of MsbA. After pulling up Sav1866 and Chang's MsbA from *S. typhimurium* on a computer screen, Locher says he realized in minutes that the MsbA structure was inverted. Interpreting the "hand" of a molecule is always a challenge for crystallographers,

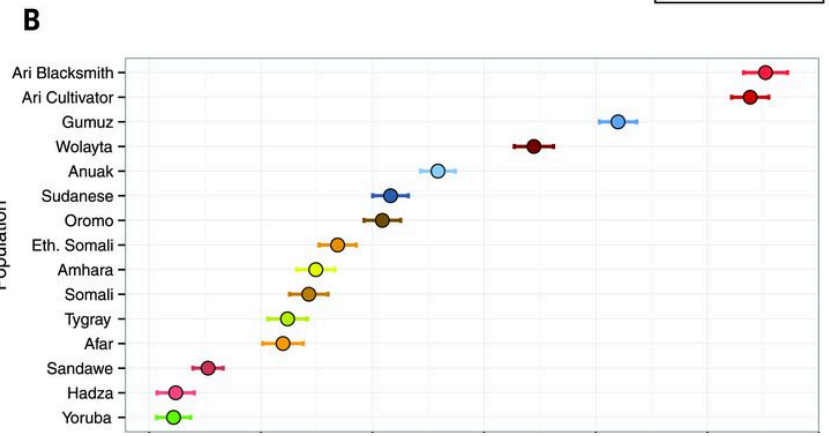
"Chang's data are good… but the faulty software threw everything off"

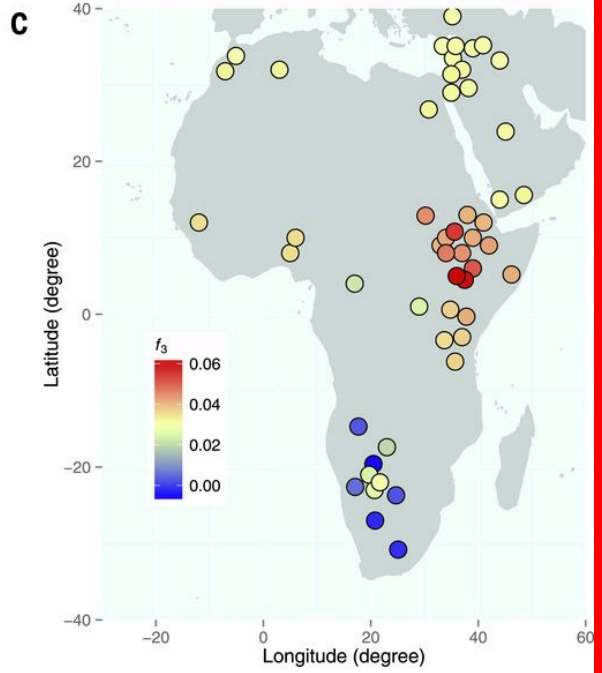
"a homemade data-analysis program had flipped two columns"

www.software.ac.uk

http://science.sciencemag.org/content/314/5807/1856.full





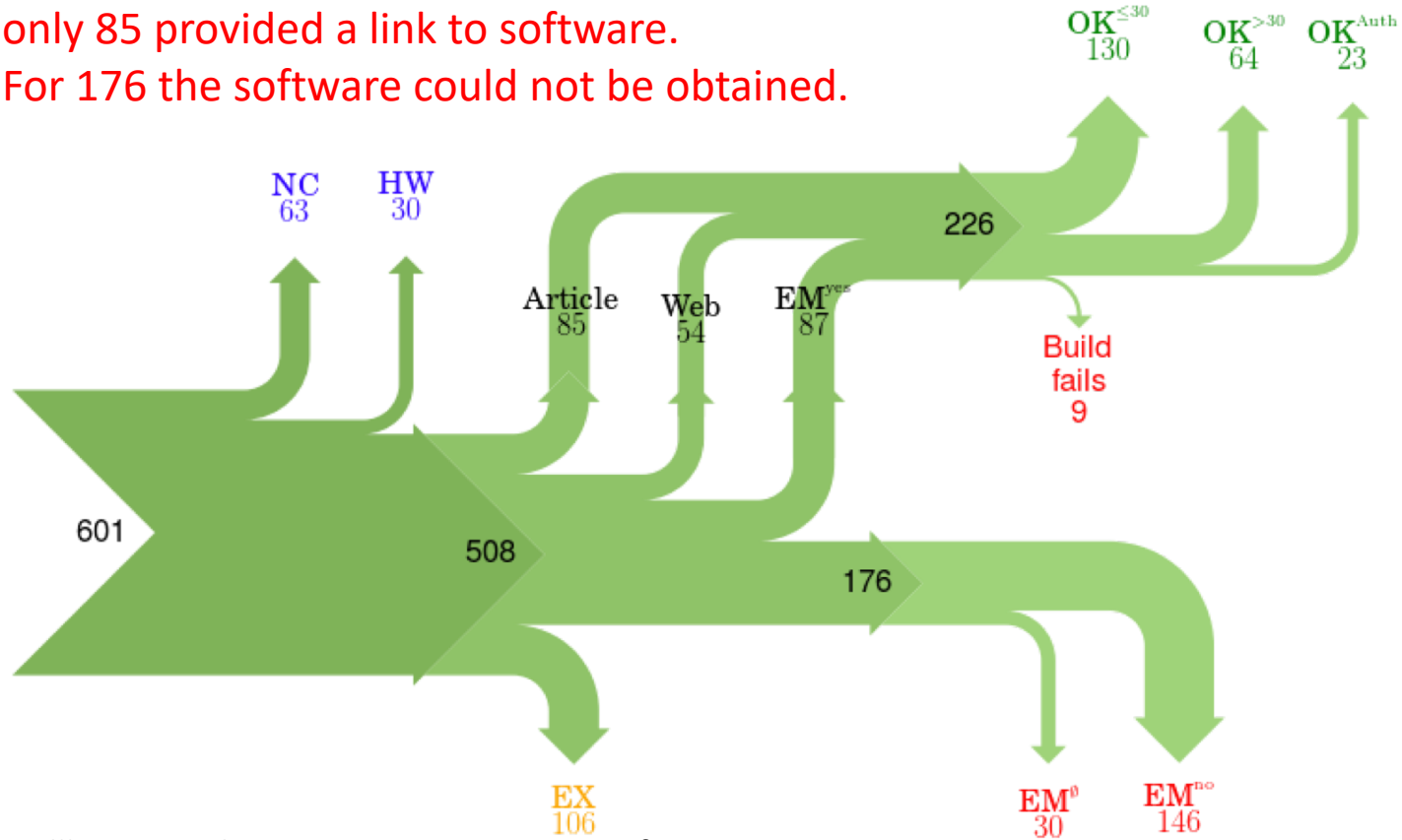


Llorente et al. Science, 350, 6262
doi:10.1126/science.aad2879

The results presented in the Report “Ancient Ethiopian genome reveals extensive Eurasian admixture throughout the African continent“ were affected by a bioinformatics error

www.software.ac.uk

Of 601 papers in ACM Computer Science journals and proceedings, only 85 provided a link to software.
For 176 the software could not be obtained.

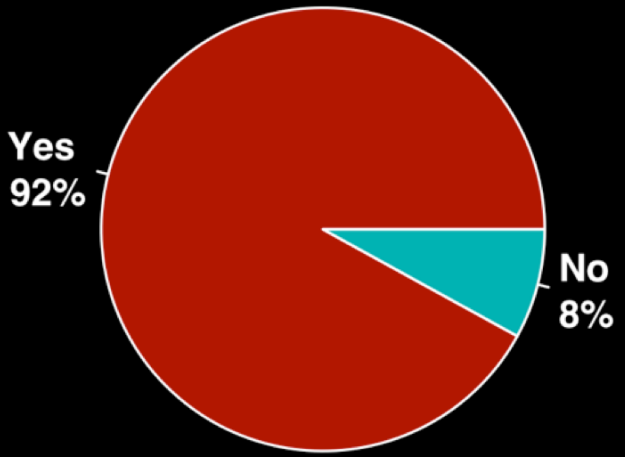


Collberg, Proebsting, Warren, University of Arizona TR 14-04, 2015
http://reproducibility.cs.arizona.edu/v2/RepeatabilityTR.pdf

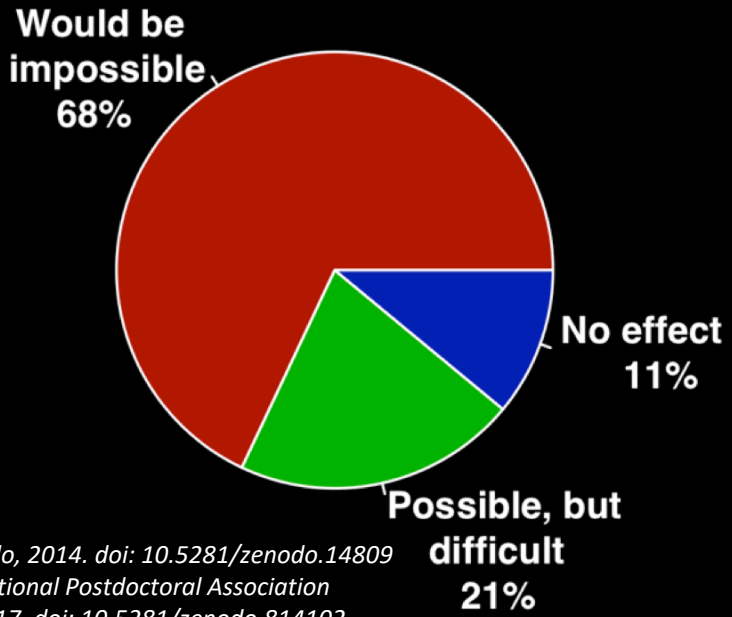
Research depends on software

www.software.ac.uk

Do you use research software?



What would happen to your research without software



56%

Of UK researchers develop their own research software

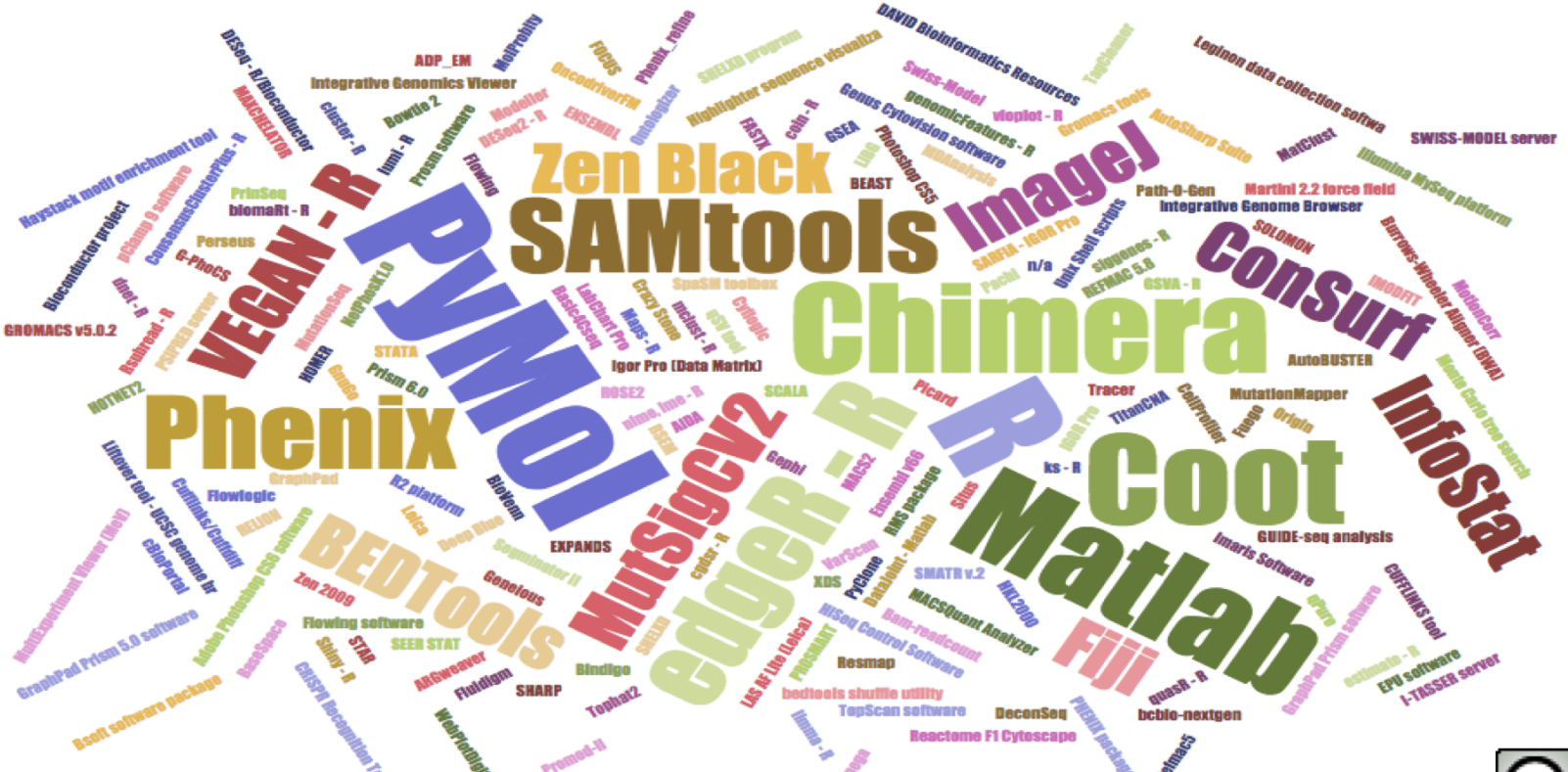
71%

Of UK researchers have had no formal software development training

S.J. Hettrick,, et al, "UK Research Software Survey 2014", Zenodo, 2014. doi: 10.5281/zenodo.14809
U. Nangia and D. S. Katz, "Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research," Zenodo, 2017. doi: 10.5281/zenodo.814102

Software in nature

www.software.ac.uk



Nangia and Katz: https://arxiv.org/pdf/1706.06527.pdf

If software is so important, why do researchers find it so difficult?

Barriers to Data and Code Sharing in Computational Science

Survey of Machine Learning Community, NIPS (Stodden, 2010):

Code		Data
77%	Time to document and clean up	54%
52%	Dealing with questions from users	34%
44%	Not receiving attribution	42%
40%	Possibility of patents	-
34%	Legal Barriers (ie. copyright)	41%
-	Time to verify release with admin	38%
30%	Potential loss of future publications	35%
30%	Competitors may get an advantage	33%
20%	Web/disk space limitations	29%

Victoria Stodden, AMP 2011 http://www.stodden.net/AMP2011/,
Special Issue Reproducible Research Computing in Science and Engineering July/August 2012, 14(4)
Howison and Herbsleb (2013) "Incentives and Integration In Scientific Software Production" CSCW 2013.

It’s still all about reputation

“This particular project was something I wrote a couple years ago to help me out with a workflow… I’d put it up on Github, so that others could potentially use it or use the code. So I went to see what people were saying about this project. It seemed liked I’d done something fundamentally wrong, so stupid that it flabbergasts someone... So of course I start sobbing. Then I see these people’s follower count, and I sob harder. I can’t help but think of potential future employers that are no longer potential.”

http://www.software.ac.uk/blog/2013-01-25-haters-gonna-hate-why-you-shouldnt-be-ashamed-releasing-your-code

Five steps to better research

1. **Train yourself and your team**
2. Write for strangers
3. Develop a Software Management Plan
4. Publish your code
5. Get expert help

Get some training

www.software.ac.uk

admin@software-carpentry.org

Teach basic lab skills for scientific computing so that researchers can do more in less time and with less pain.

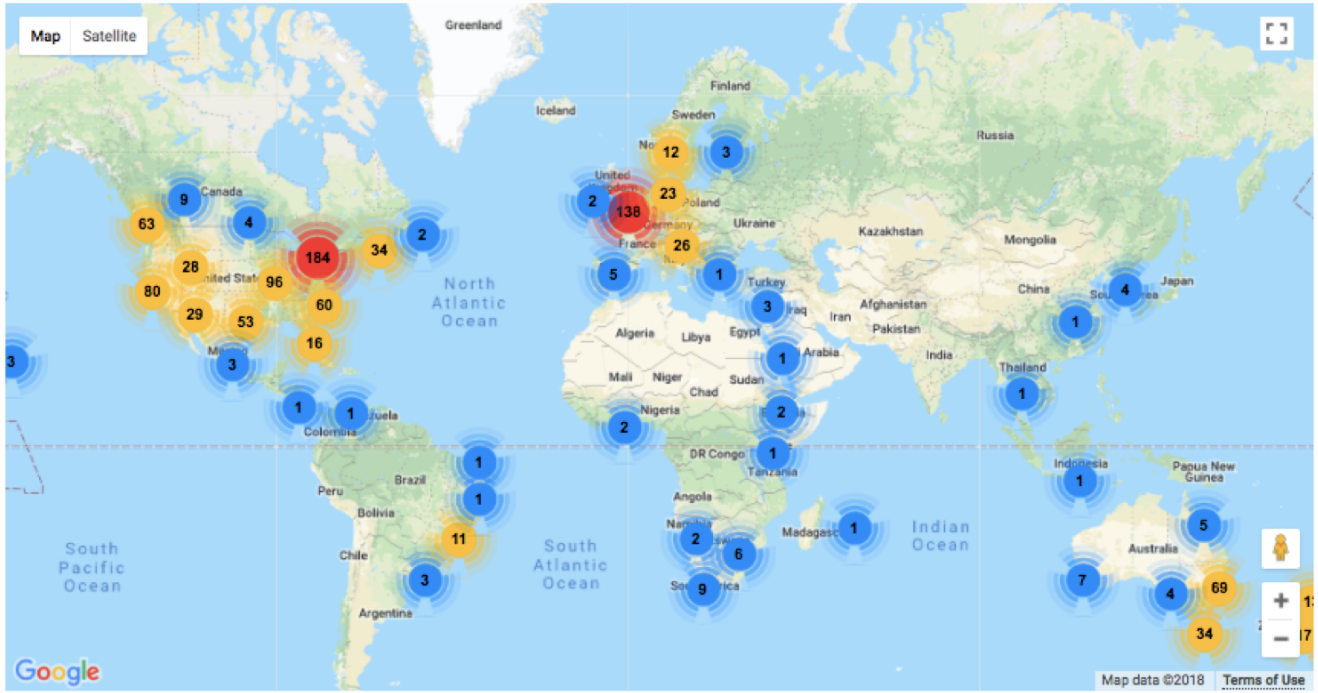
admin@datacarpentry.org

Teach basic concepts, skills and tools for working more effectively with data. Workshops are designed for people with little to no prior computational experience.

Open source learning, that can be tailored to disciplines.
"Train the trainers": building a capable base of instructors.

A global community of practice

www.software.ac.uk



Five steps to better research

1. Train yourself and your team
2. **Write for strangers**
3. Develop a Software Management Plan
4. Publish your code
5. Get expert help

Writing for strangers

- “In pursuing full reproducibility, we fundamentally produce code for strangers to use. Although it might seem unnatural to help strangers—and reluctance to do so certainly impedes the spread of reproducible research—our term *stranger* really means anyone who doesn’t possess our current short-term memory and experiences.”
  – David Donoho

Definition of a stranger

- Anyone not in possession of the authors current short-term memory and experiences i.e.
  - People reading your paper
  - Future research (and industry) collaborators
  - Referees of your paper / grant proposal
  - Future members of your research group
  - Current students
  - Co-authors
  - You, the author, in six months time

OPEN ACCESS Freely available online

PLOS | BIOLOGY

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Guy[6¤], Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11], Ethan P. White[12], Paul Wilson[13]

1 Mozilla Foundation, Toronto, Ontario, Canada, **2** University of Ontario Institute of Technology, Oshawa, Ontario, Canada, **3** Michigan State University, East Lansing, Michigan, United States of America, **4** Software Sustainability Institute, Edinburgh, United Kingdom, **5** Space Telescope Science Institute, Baltimore, Maryland, United States of America, **6** University of Toronto, Toronto, Ontario, Canada, **7** Monterey Bay Aquarium Research Institute, Moss Landing, California, United States of America, **8** University of California Berkeley, Berkeley, California, United States of America, **9** University of British Columbia, Vancouver, British Columbia, Canada, **10** Queen Mary University of London, London, United Kingdom, **11** University College London, London, United Kingdom, **12** Utah State University, Logan, Utah, United States of America, **13** University of Wisconsin, Madison, Wisconsin, United States of America

Introduction

Scientists spend an increasing amount of time building and using software. However, most scientists are never taught how to do this efficiently. As a result, many are unaware of tools and practices that would allow them to write more reliable and maintainable code with less effort. We describe a set of best practices for scientific software development that have solid foundations in research and experience, and that improve scientists' productivity and the reliability of their software.

Software is as important to modern scientific research as telescopes and test tubes. From groups that work exclusively on computational problems, to traditional laboratory and field scientists, more and more of the daily operation of science revolves around developing new algorithms, managing and analyzing the large amounts of data that are generated in single research projects, combining disparate datasets to assess synthetic problems, and other computational tasks.

Scientists typically develop their own software for these purposes because doing so requires substantial domain-specific knowledge. As a result, recent studies have found that scientists typically spend 30% or more of their time developing software [1,2]. However, 90% or more of them are primarily self-taught [1,2], and therefore lack exposure to basic software development practices such as writing maintainable code, using version control and issue trackers, code reviews, unit testing, and task automation.

We believe that software is just another kind of experimental apparatus [3] and should be built, checked, and used as carefully as any physical apparatus. However, while most scientists are careful to validate their laboratory and field equipment, most do not know how reliable their software is [4,5]. This can lead to serious errors impacting the central conclusions of published research [6]: recent high-profile retractions, technical comments, and corrections because of errors in computational methods include papers in *Science* [7,8], *PNAS* [9], the *Journal of Molecular Biology* [10], *Ecology Letters* [11,12], the *Journal of Mammalogy* [13], *Journal of the American College of Cardiology* [14], *Hypertension* [15], and *The American Economic Review* [16].

In addition, because software is often used for more than a single project, and is often reused by other scientists, computing errors can have disproportionate impacts on the scientific process. This type of cascading impact caused several prominent retractions when an error from another group's code was not discovered until after publication [6]. As with bench experiments, not everything must be done to the most exacting standards; however, scientists need to be aware of best practices both to improve their own approaches and for reviewing computational work by others.

This paper describes a set of practices that are easy to adopt and have proven effective in many research settings. Our recommendations are based on several decades of collective experience both building scientific software and teaching computing to scientists [17,18], reports from many other groups [19–25], guidelines for commercial and open source software development [26,27], and on empirical studies of scientific computing [28–31] and software development in general (summarized in [32]). None of these practices will guarantee efficient, error-free software development, but used in concert they will reduce the number of errors in scientific software, make it easier to reuse, and save the authors of the software time and effort that can used for focusing on the underlying scientific questions.

Our practices are summarized in Box 1; labels in the main text such as "(1a)" refer to items in that summary. For reasons of space, we do not discuss the equally important (but independent) issues of reproducible research, publication and citation of code and data, and open science. We do believe, however, that all of these will be much easier to implement if scientists have the skills we describe.

Citation: Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745

Academic Editor: Jonathan A. Eisen, University of California Davis, United States of America

Published January 7, 2014

Funding: Neil Chue Hong was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) Grant EP/H043160/1 for the UK Software Sustainability Institute. Ian M. Mitchell was supported by NSERC Discovery Grant #298211. Mark Plumbley was supported by EPSRC through a Leadership Fellowship (EP/G007144/1) and a grant (EP/H043101/1) for SoundSoftware.ac.uk. Ethan White was supported by a CAREER grant from the US National Science Foundation (DEB 0953694). Greg Wilson was supported by a grant from the Sloan Foundation. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The lead author (GVW) is involved in a pilot study of code review in scientific computing with *PLOS Computational Biology*.

* E-mail: gvwilson@software-carpentry.org

¤ **Current address:** Microsoft, Inc., Seattle, Washington, United States of America

The Community Page is a forum for organizations and societies to highlight their efforts to enhance the dissemination and value of scientific knowledge.

Box 1. Summary of Best Practices

1. Write programs for people, not computers.
   (a) A program should not require its readers to hold more than a handful of facts in memory at once.
   (b) Make names consistent, distinctive, and meaningful.
   (c) Make code style and formatting consistent.
2. Let the computer do the work.
   (a) Make the computer repeat tasks.
   (b) Save recent commands in a file for re-use.
   (c) Use a build tool to automate workflows.
3. Make incremental changes.
   (a) Work in small steps with frequent feedback and course correction.
   (b) Use a version control system.
   (c) Put everything that has been created manually in version control.
4. Don't repeat yourself (or others).
   (a) Every piece of data must have a single authoritative representation in the system.
   (b) Modularize code rather than copying and pasting.
   (c) Re-use code instead of rewriting it.
5. Plan for mistakes.
   (a) Add assertions to programs to check their operation.
   (b) Use an off-the-shelf unit testing library.
   (c) Turn bugs into test cases.
   (d) Use a symbolic debugger.
6. Optimize software only after it works correctly.
   (a) Use a profiler to identify bottlenecks.
   (b) Write code in the highest-level language possible.
7. Document design and purpose, not mechanics.
   (a) Document interfaces and reasons, not implementations.
   (b) Refactor code in preference to explaining how it works.
   (c) Embed the documentation for a piece of software in that software.
8. Collaborate.
   (a) Use pre-merge code reviews.
   (b) Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
   (c) Use an issue tracking tool.

Write Programs for People, Not Computers

Scientists writing software need to write code that both executes correctly and can be easily read and understood by other programmers (especially the author's future self). If software cannot be easily read and understood, it is much more difficult to know that it is actually doing what it is intended to do. To be productive, software developers must therefore take several aspects of human cognition into account: in particular, that human working memory is limited, human pattern matching abilities are finely tuned, and human attention span is short [33–37].

First, ***a program should not require its readers to hold more than a handful of facts in memory at once (1a)***. Human working memory can hold only a handful of items at a time, where each item is either a single fact or a "chunk" aggregating several facts [33,34], so programs should limit the total number of items to be remembered to accomplish a task. The primary way to accomplish this is to break programs up into easily understood functions, each of which conducts a single, easily understood, task. This serves to make each piece of the program easier to understand in the same way that breaking up a scientific paper using sections and paragraphs makes it easier to read.

Second, scientists should ***make names consistent, distinctive, and meaningful (1b)***. For example, using non-descriptive names, like a and foo, or names that are very similar, like results and results2, is likely to cause confusion.

Third, scientists should ***make code style and formatting consistent (1c)***. If different parts of a scientific paper used different formatting and capitalization, it would make that paper more difficult to read. Likewise, if different parts of a program are indented differently, or if programmers mix CamelCaseNaming and pothole_case_naming, code takes longer to read and readers make more mistakes [35,36].

Let the Computer Do the Work

Science often involves repetition of computational tasks such as processing large numbers of data files in the same way or regenerating figures each time new data are added to an existing analysis. Computers were invented to do these kinds of repetitive tasks but, even today, many scientists type the same commands in over and over again or click the same buttons repeatedly [17]. In addition to wasting time, sooner or later even the most careful researcher will lose focus while doing this and make mistakes.

Scientists should therefore ***make the computer repeat tasks (2a)*** and ***save recent commands in a file for re-use (2b)***. For example, most command-line tools have a "history" option that lets users display and re-execute recent commands, with minor edits to filenames or parameters. This is often cited as one reason command-line interfaces remain popular [38,39]: "do this again" saves time and reduces errors.

A file containing commands for an interactive system is often called a *script*, though there is real no difference between this and a program. When these scripts are repeatedly used in the same way, or in combination, a workflow management tool can be used. The paradigmatic example is compiling and linking programs in languages such as Fortran, C++, Java, and C# [40]. The most widely used tool for this task is probably Make (http://www.gnu.org/software/make), although many alternatives are now available [41]. All of these allow people to express dependencies between files, i.e., to say that if A or B has changed, then C needs to be updated using a specific set of commands. These tools have been successfully adopted for scientific workflows as well [42].

To avoid errors and inefficiencies from repeating commands manually, we recommend that scientists ***use a build tool to automate workflows (2c)***, e.g., specify the ways in which intermediate data files and final results depend on each other, and on the programs that create them, so that a single command will regenerate anything that needs to be regenerated.

Best Practices for Scientific Computing: https://doi.org/10.1371/journal.pbio.1001745

Good Enough Practices in Scientific Computing

Greg Wilson[1,‡*], Jennifer Bryan[2,‡], Karen Cranston[3,‡], Justin Kitzes[4,‡], Lex Nederbragt[5,‡], Tracy K. Teal[6,‡]
1 Software Carpentry Foundation / gvwilson@software-carpentry.org
2 University of British Columbia / jenny@stat.ubc.ca
3 Duke University / karen.cranston@duke.edu
4 University of California, Berkeley / jkitzes@berkeley.edu
5 University of Oslo / lex.nederbragt@ibv.uio.no
6 Data Carpentry / tkteal@datacarpentry.org

‡ These authors contributed equally to this work.
* E-mail: Corresponding gvwilson@software-carpentry.org

Abstract

We present a set of computing tools and techniques that every researcher can and should adopt. These recommendations synthesize inspiration from our own work, from the experiences of the thousands of people who have taken part in Software Carpentry and Data Carpentry workshops over the past six years, and from a variety of other guides. Our recommendations are aimed specifically at people who are new to research computing.

Author Summary

Computers are now essential in all branches of science, but most researchers are never taught the equivalent of basic lab skills for research computing. As a result, they take days or weeks to do things that could be done in minutes or hours, are often unable to reproduce their own work (much less the work of others), and have no idea how reliable their computational results are.

This paper presents a set of good computing practices that every researcher can adopt regardless of their current level of technical skill. These practices, which encompass data management, programming, collaborating with colleagues, organizing projects, tracking work, and writing manuscripts, are drawn from a wide variety of published sources, from our daily lives, and from our work with volunteer organizations that have delivered workshops to over 11,000 people since 2010.

Introduction

Two years ago a group of researchers involved in Software Carpentry[1] and Data Carpentry[2] wrote a paper called "Best Practices for Scientific Computing" [1]. It was well received, but many novices found its litany of tools and techniques intimidating. Also, by definition, the "best" are a small minority. What practices are comfortably within reach for the "rest"?

[1] http://software-carpentry.org/

Box 1: Summary of Practices

1. Data Management
   a) Save the raw data.
   b) Create the data you wish to see in the world.
   c) Create analysis-friendly data.
   d) Record all the steps used to process data.
   e) Anticipate the need to use multiple tables.
   f) Submit data to a reputable DOI-issuing repository so that others can access and cite it.
2. Software
   a) Place a brief explanatory comment at the start of every program.
   b) Decompose programs into functions.
   c) Be ruthless about eliminating duplication.
   d) Always search for well-maintained software libraries that do what you need.
   e) Test libraries before relying on them.
   f) Give functions and variables meaningful names.
   g) Make dependencies and requirements explicit.
   h) Do not comment and uncomment sections of code to control a program's behavior.
   i) Provide a simple example or test data set.
   j) Submit code to a reputable DOI-issuing repository.
3. Collaboration
   a) Create an overview of your project.
   b) Create a shared public "to-do" list.
   c) Make the license explicit.
   d) Make the project citable.
4. Project Organization
   a) Put each project in its own directory, which is named after the project.
   b) Put text documents associated with the project in the `doc` directory.
   c) Put raw data and metadata in a `data` directory, and files generated during cleanup and analysis in a `results` directory.
   d) Put project source code in the `src` directory.
   e) Put external scripts, or compiled programs in the `bin` directory.
   f) Name all files to reflect their content or function.
5. Keeping Track of Changes
   a) Back up (almost) everything created by a human being as soon as it is created.
   b) Keep changes small.
   c) Share changes frequently.
   d) Create, maintain, and use a checklist for saving and sharing changes to the project.
   e) Store each project in a folder that is mirrored off the researcher's working machine.
   f) Use a file called `CHANGELOG.txt` to record changes, and
   g) Copy the entire project whenever a significant change has been made, OR
   h) Use a version control system to manage changes
6. Manuscripts
   a) Write manuscripts using online tools with rich formatting, change tracking, and reference management, OR
   b) Write the manuscript in a plain text format that permits version control

Good Enough Practices in Scientific Computing: https://doi.org/10.1371/journal.pcbi.1005510

Good Enough Practices - Data

- Data – play FAIR:
  - Save and backup raw data
  - Create analysis-friendly data
  - Record your processing steps
  - Anticipate the need to use multiple tables, and use a unique identifier for each record
  - Submit data to a repository and get a DOI

Good Enough Practices - Software

- Use version control
- Document for your future self
- Learn to be modular
- Make it accessible in the future; choose a license
- Get a colleague to try using it
- Ask a collaborator to contribute and give feedback
- *Automate your process*

Community standards

- ESIP (Earth Sciences): https://esipfed.github.io/Software-Assessment-Guidelines/
- CLARIAH (Arts and Humanities): https://github.com/CLARIAH/software-quality-guidelines
- IPOL (Image Processing): https://tools.ipol.im/wiki/ref/software_guidelines/
- ELIXIR (Life Sciences): https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5490478/
- *Your community here?*

Five steps to better research

1. Train yourself and your team
2. Write for strangers
3. **Develop a Software Management Plan**
4. Publish your code
5. Get expert help

Software Management Plans

- Much research software development is not formally planned
  - Developed to solve a research question
  - Evolved rather than planned
  - Even larger research software projects tend to be driven by a single person to start
- Software Management Plans are a way of thinking through the process of running a research software development project

How do I write a plan?

- The Software Sustainability Institute has drawn up a checklist
  - http://www.software.ac.uk/software-management-plans
- Series of questions to help you consider all aspects of the development of your software
- Complementary to Data Management Plans
  - https://dmponline.dcc.ac.uk/

A Minimal Plan

- What software will you write?
  - What will your software do?
  - Will your software have a name?
- Who are the intended users of your software?
  - Is for one type of user or for many?
  - What expertise is required?
- How will you make your software available?
- How will your software contribute to research and how will you measure its contribution?

Five steps to better research

1. Train yourself and your team
2. Write for strangers
3. Develop a Software Management Plan
4. **Publish your code**
5. Get expert help

Sharing is key to reproducibility

- Improves transparency
- Improves understanding
- Elimination of errors
- Encourages collaboration
- Easier on-ramping

- Improves trust

“Deep intellectual contributions now encoded only in software" – Stodden

“Scholarship is the full software environment, code and data, that produced the result” - Claerbout

Research Software Workflow

www.software.ac.uk

describe

GitLab

GitHub

Bitbucket

<sword />

zenodo

DRYAD

figshare

credit for all your research

develop → share → preserve

Jenkins

Travis CI

Buildbot

docker

VAGRANT

Developed and versioned using code repository

Published via code repository or website

Deposited in digital repository with paper / for preservation

Get and give credit for software

- Mechanisms
  - Software papers http://bit.ly/softwarejournals
  - Software citation e.g. Software Citation Working Group https://www.force11.org/group/software-citation-working-group
- Tools
  - Researcher Identifiers e.g. ORCID http://orcid.org/
  - Alt-Metrics e.g. ImpactStory http://impactstory.org/
- Be a better reviewer
  - Ask to see code and data
  - Be constructive in your criticism

Literate Programming

- Traditional papers are just advertisements
  - A literate computing document is the research
- The technology is out there
  - Jupyter notebooks
  - Mathematica
  - R Markdown
  - knitR
  - MATLAB Live scripts

LIGO Example

www.software.ac.uk

PRL 116, 061102 (2016) | Selected for a Viewpoint in *Physics* | PHYSICAL REVIEW LETTERS | week ending 12 FEBRUARY 2016

Observation of Gravitational Waves from a Binary Black Hole Merger

B. P. Abbott *et al.*[*]
(LIGO Scientific Collaboration and Virgo Collaboration)
(Received 21 January 2016; published 11 February 2016)

On September 14, 2015 at 09:50:45 UTC the two detectors of the Laser Interferometer Gravitational-Wave Observatory simultaneously observed a transient gravitational-wave signal. The signal sweeps upwards in frequency from 35 to 250 Hz with a peak gravitational-wave strain of 1.0×10^{-21}. It matches the waveform predicted by general relativity for the inspiral and merger of a pair of black holes and the ringdown of the resulting single black hole. The signal was observed with a matched-filter signal-to-noise ratio of 24 and a false alarm rate estimated to be less than 1 event per 203 000 years, equivalent to a significance greater than 5.1σ. The source lies at a luminosity distance of 410^{+160}_{-180} Mpc corresponding to a redshift $z = 0.09^{+0.03}_{-0.04}$. In the source frame, the initial black hole masses are $36^{+5}_{-4}M_{\odot}$ and $29^{+4}_{-4}M_{\odot}$, and the final black hole mass is $62^{+4}_{-4}M_{\odot}$, with $3.0^{+0.5}_{-0.5}M_{\odot}c^2$ radiated in gravitational waves. All uncertainties define 90% credible intervals. These observations demonstrate the existence of binary stellar-mass black hole systems. This is the first direct detection of gravitational waves and the first observation of a binary black hole merger.

DOI: 10.1103/PhysRevLett.116.061102

I. INTRODUCTION

In 1916, the year after the final formulation of the field equations of general relativity, Albert Einstein predicted the existence of gravitational waves. He found that the linearized weak-field equations had wave solutions: transverse waves of spatial strain that travel at the speed of light, generated by time variations of the mass quadrupole

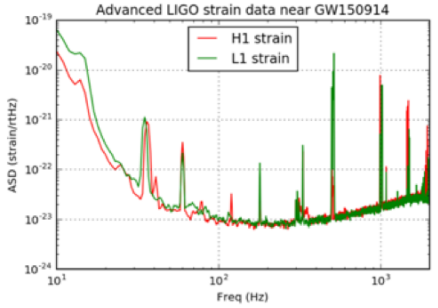
The discovery of the binary pulsar system PSR B1913+16 by Hulse and Taylor [20] and subsequent observations of its energy loss by Taylor and Weisberg [21] demonstrated the existence of gravitational waves. This discovery, along with emerging astrophysical understanding [22], led to the recognition that direct observations of the amplitude and phase of gravitational waves would enable

There's a signal in these data! For the moment, let's ignore that, and assume it's all noise.

In [7]:
```
# number of sample for the fast fourier transform:
NFFT = 1*fs
fmin = 10
fmax = 2000
Pxx_H1, freqs = mlab.psd(strain_H1, Fs = fs, NFFT = NFFT)
Pxx_L1, freqs = mlab.psd(strain_L1, Fs = fs, NFFT = NFFT)

# We will use interpolations of the ASDs computed above for whitening:
psd_H1 = interp1d(freqs, Pxx_H1)
psd_L1 = interp1d(freqs, Pxx_L1)

# plot the ASDs:
plt.figure()
plt.loglog(freqs, np.sqrt(Pxx_H1),'r',label='H1 strain')
plt.loglog(freqs, np.sqrt(Pxx_L1),'g',label='L1 strain')
plt.axis([fmin, fmax, 1e-24, 1e-19])
plt.grid('on')
plt.ylabel('ASD (strain/rtHz)')
plt.xlabel('Freq (Hz)')
plt.legend(loc='upper center')
plt.title('Advanced LIGO strain data near GW150914')
plt.savefig('GW150914_ASDs.png')
```



NOTE that we only plot the data between fmin = 10 Hz and fmax = 2000 Hz.

Five steps to better research

1. Train yourself and your team
2. Write for strangers
3. Develop a Software Management Plan
4. Publish your code
5. **Get expert help**

Research Software Engineers

www.software.ac.uk

Software Sustainability Institute

About | Blog | Community | Policy | Software | Training | Resources

Launching the German RSE chapter "de-RSE"

By Martin Hammitzsch, GFZ Potsdam, Stephan Janosch, MPI CBG & Frank Loeffler, Louisiana State University

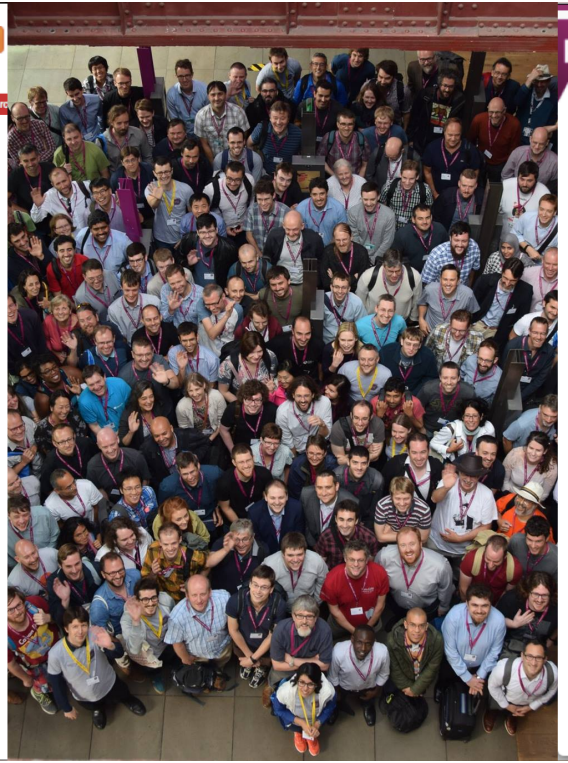
Founders of the de-RSE chapter

The days following the first conference of Research Software Engineers (RSEs) saw the launch of a German RSE chapter *de-RSE*. It was formed by RSEs working inside and outside Germany, and it will further the shared objectives of RSEs and become the collective mouthpiece for RSEs within the German science.

All of the authors were exposed to the day-to-day problems caused by using software in science, and this meant that many of us were following the Software Sustainability Institute, and a few other activities around the globe. The lucky ones among us were even able to participate in events over the last few years to see how to improve our situation. Over the last few years a critical mass of motivated Research Software Engineers (RSEs) formed at various locations across Europe, North America and a few other countries. Then in September 2016, the world's first conference for RSEs took place in Manchester. It was the right time for this event. Bringing together RSEs lead to discussions about how to transfer the UKRSE spirit to other countries. How could other national science systems benefit from the professionalisation of software engineering in sciences? How can the people behind research software receive the the acknowledgement and resources they deserve?

Our contribution is mainly, but not only, in the area of scientific software development. We want to prepare science for the ever-increasing influence of new technologies and new IT concepts, help the community cope with the challenges posed by the increasing digitalisation of research, and help to exploit the resulting opportunities.

We believe that the de-RSE can promote the sustainability and falsifiability of scientific software development as part of research processes, persuade more researchers to view software as a foundation of publications and research data, and promote scientific software as a key building block in Open Science.



RSE de-RSE

Research Software Engineers (RSEs) - verantwortlich für wissenschaftliche Software

Softwareentwicklung ist ein elementarer, unverzichtbarer Bestandteil der Forschungstätigkeit. Wissenschaftliche Software unterstützt zunehmend die Gewinnung, Verarbeitung und Auswertung von empirischen Daten, aber auch die Modellierung und Simulation von komplexen Prozessen. Damit hat Software einen maßgeblichen Einfluss auf die Qualität der erzielten Forschungsergebnisse. Das Britische Software Sustainability Institut (SSI) hat dafür den Slogan "Better Software - Better Research" geprägt.

Jedoch spiegelt der aktuelle Umgang mit Software und die Bewertung der Software-Entwicklung die Bedeutung dieser Arbeit im Forschungsprozess nicht in angemessener Weise wider. Die Ursachen dafür sind breit gefächert und äußern sich in vielfältigen Befunden, zum Beispiel:

- Fehlende Anerkennung für Software-Entwicklung als wissenschaftliche Leistung
- Fehlende Verankerung im wissenschaftlichen Reputationssystem
- Eingeschränkte Verfügbarkeit und Nutzbarkeit wissenschaftlicher Software
- Mehrfach- und Parallelentwicklung
- Ungenügende Kompetenzen im Software-Engineering
- Fehlende Qualitätsstandards für die Entwicklung und den Review wissenschaftlicher Software
- Fehlende Reproduzierbarkeit, z.B. von Simulationsergebnissen
- Unklare Regelungen zur Veröffentlichung in Bezug auf Lizenzen und Intellectual Property (IP)

In der Forschung tätige Informatiker, Wissenschaftler und andere Softwareentwickler sind in der RSE-Gemeinschaft zusammen gekommen und haben es sich zur Aufgabe gemacht, dieses Problem deutlich sichtbar zu machen und die gegenwärtige Situation entscheidend zu verbessern.

www.rse.ac.uk Software Sustainability Institute **www.de-rse.org**

www.software.ac.uk

Don’t be afraid to let others see your software

~~Five~~ Six steps to better research

www.software.ac.uk

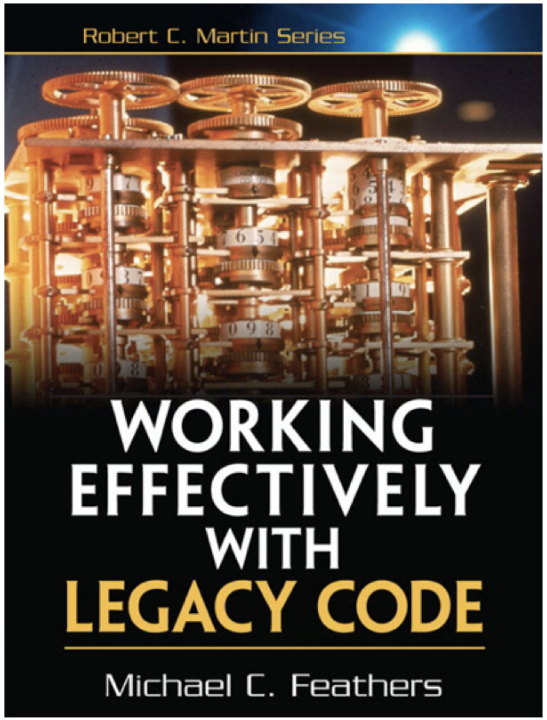
1. Train yourself and your team
2. Write for strangers
3. Develop a Software Management Plan
4. Publish your code
5. Get expert help
6. **Keep improving, incrementally**

Slides: https://doi.org/10.6084/m9.figshare.5930662

All software is “legacy code”

“Many of us have tried to discover ways to prevent code from becoming legacy. But … prevention is imperfect. Even the most disciplined development team, knowing the best principles, using the best patterns, and following the best practices will create messes from time to time. The rot still accumulates. It’s not enough to prevent the rot – you have to be able to reverse it.”

Technical debt will happen – understand how to make your payments!



Learning from Open Source

www.software.ac.uk

producingoss.com

- Scientific software development projects share characteristics with open source software projects
  - But more research has been done on OSS projects
- Karl Fogel's *"Producing Open Source Software: How to Run a Successful Free Software Project"* distills this knowledge into a practical guide

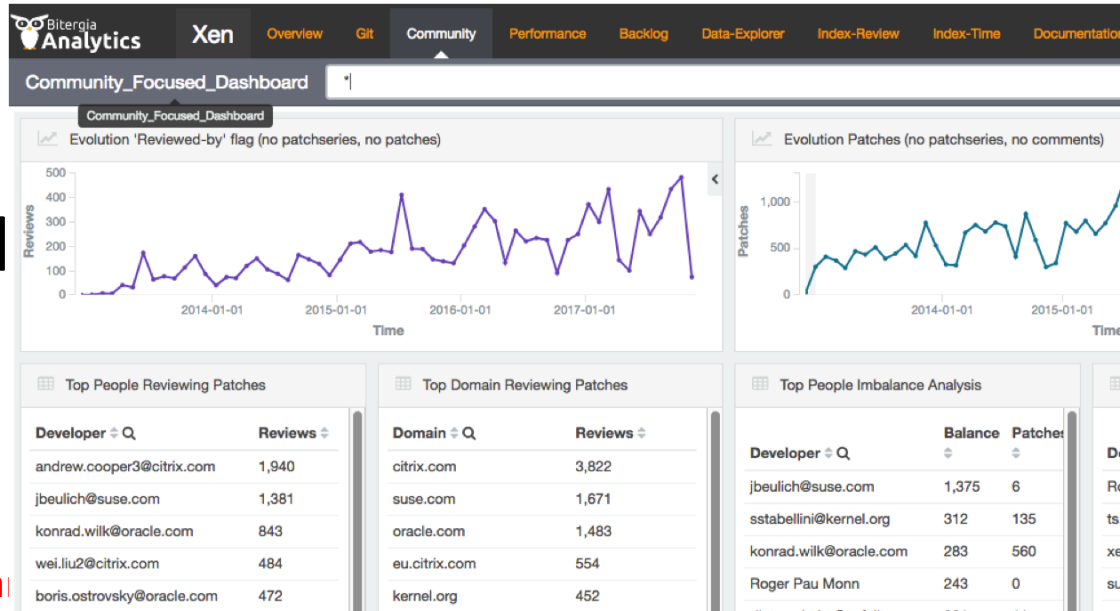
Measuring project health

www.software.ac.uk

- As your project is running, how do you understand if it's going well?
- Metrics can be used to assess progress

CHAOSS

https://chaoss.community/



Other resources

- Software sustainability evaluation
  - https://www.software.ac.uk/online-sustainability-evaluation
- Software Sustainability Institute resources
  - Guides: https://www.software.ac.uk/guides
  - Top Tips: https://www.software.ac.uk/resources/top-tips
- Scientific Software Practice
  - Good Enough Practices in Scientific Computing: https://doi.org/10.1371/journal.pcbi.1005510
  - Best Practices for Scientific Computing: https://doi.org/10.1371/journal.pbio.1001745
- Software Citation
  - Principles: https://peerj.com/articles/cs-86/
  - GitHub – Zenodo: https://guides.github.com/activities/citable-code/
- RDA Active Data Management Plans IG
  - https://www.rd-alliance.org/groups/active-data-management-plans.html

Find out more about the SSI

www.software.ac.uk

- Community Engagement (Lead: Shoaib Sufi)
  - Fellowship Programme & Events and Workshops
- Consultancy (Lead: Steve Crouch)
  - Open Call for Projects / Collaborations
  - Online Software Evaluation & Software Management Planning
- Policy and Publicity (Lead: Simon Hettrick)
  - Case Studies / Policy Campaigns
  - Software and Research Blog
- Training (Lead: Aleksandra Nenadic)
  - Software Carpentry and Data Carpentry
  - Guides and Top Tips
- Journal of Open Research Software (Editor: Neil Chue Hong)

Collaboration between universities of Edinburgh, Manchester, Oxford and Southampton
Supported by EPSRC Grant EP/H043160/1 + EPSRC/ESRC/BBSRC grant EP/N006410/1

Six steps to better research

1. Train yourself and your team
2. Write for strangers
3. Develop a Software Management Plan
4. Publish your code
5. Get expert help
6. Keep improving, incrementally

Slides: https://doi.org/10.6084/m9.figshare.5930662

Acknowledgements

The SSI team/*alumni*:
- Aleksandra Nenadic
- *Aleksandra Pawlik*
- *Alexander Hay*
- *Arno Proeme*
- Carole Goble
- Claire Wyatt
- Clem Hadfield
- Dave De Roure
- *Devasena Prasad*
- Giacomo Peru
- Graeme Smith
- *Iain Emsley*
- James Graham
- John Robinson
- Les Carr
- *Malcolm Atkinson*
- *Malcolm Illingworth*
- Mario Antonioletti
- Mark Parsons
- Mike Jackson
- Olivier Philippe
- *Priyanka Singh*
- Raniere Silva
- *Rob Baxter*
- *Robin Wilson*
- Shoaib Sufi
- Simon Hettrick
- Stephen Crouch
- *Tim Parkinson*
- *Toni Collis*
- *Plus the SSI Fellows and RSE community*

Scientific software:
- Dan Katz
- Heather Piowowar
- James Howison
- Jeff Carver
- Jennifer Schopf
- Kaitlin Thaney
- Martin Fenner
- Victoria Stodden
- WSSSPE community

Software/Data Carpentry
- Greg Wilson
- Jonah Duckles
- Tracy Teal
- Instructor Community

EPSRC Pioneering research

BBSRC

E·S·R·C ECONOMIC & SOCIAL RESEARCH COUNCIL

Jisc

NERC SCIENCE OF THE ENVIRONMENT

Supported by EPSRC Grant EP/H043160/1 + EPSRC/ESRC/BBSRC grant EP/N006410/1

Software Sustainability Institute

www.software.ac.uk

A national facility for cultivating better, more sustainable, research software to enable world-class research

- Software reaches boundaries in its development cycle that prevent improvement, growth and adoption
- Providing the expertise and services needed to negotiate to the next stage
- Developing the policy and tools to support the community developing and using research software

Supported by EPSRC Grant EP/H043160/1
+ EPSRC/ESRC/BBSRC grant EP/N006410/1

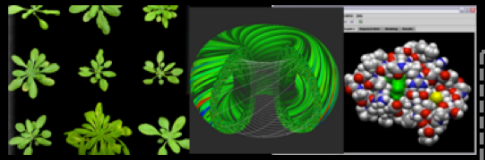
Additional slides

www.software.ac.uk

About the Institute

Software

Helping the community to develop software that meets the needs of reliable, reproducible, and reusable research



Training

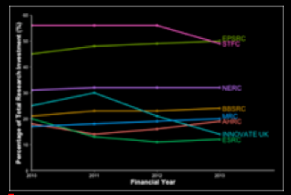
Delivering essential software skills to researchers via CDTs, institutions & doctoral schools

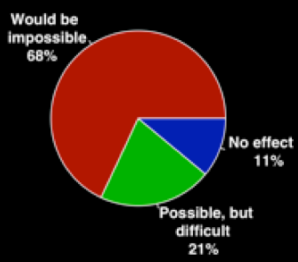
Outreach

Exploiting our platform to enable engagement, delivery & uptake

Policy

Collecting evidence on the community's software use & sharing with stakeholders

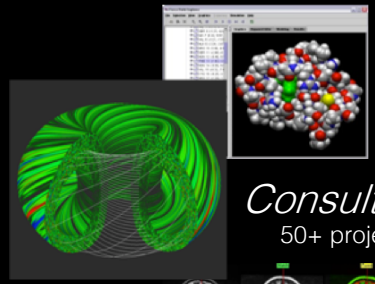




Community

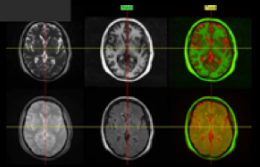
Bringing together the right people to understand and address topical issues

Software



Consultancy

50+ projects



Advice

130+ evaluations
4 surgeries

Training

Courses

35+ UK SWC workshops
1000+ learners

Guides

80+ guides
50,000 readers

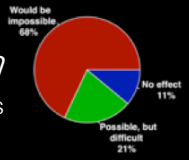
Outreach

Website & blog

150+ contributed articles
20,000 unique visitors per month
3,000 Twitter followers

Research

740 researchers
50,000 grants analysed

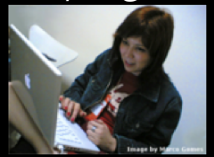


Campaigns

300+ RSEs engaged

BETTER SOFTWARE BETTER RESEARCH

2100 signatures



13 issues highlighted

Journal of open research software

Workshops

20+ workshops organised

Fellowship

61 domain ambassadors

Policy

Community

www.software.ac.uk

Software Management Plans

Software Management Plans

- Much research software development is not formally planned
  - Developed to solve a research question
  - Evolved rather than planned
  - Even larger research software projects tend to be driven by a single person to start
- Software Management Plans are a way of thinking through the process of running a research software development project

Software Management Plans

- A Software Management Plan can help:
  - Understand what and who the software is for
  - Define success criteria for the software
  - Understand what processes, resources and infrastructure are required
  - Think about the future of our software once a project or funding period ends
- A Software Management Plan is principally for a project's own use, and should be developed and agreed by the whole project team

How do I write a plan?

- The Software Sustainability Institute has drawn up a checklist
  - http://www.software.ac.uk/software-management-plans
- Series of questions to help you consider all aspects of the development of your software
- Complementary to Data Management Plans
  - https://dmponline.dcc.ac.uk/

SMP Checklist

- **About your software (who, what, why, how)**
- Software Development Infrastructure
- Developing good software
- Managing dependencies
- Managing software development
- Engaging users
- Intellectual property, copyright and licencing
- Preserving your software

A Minimal Software Management Plan

- What software will you write?
  - What will your software do?
  - Will your software have a name?
- Who are the intended users of your software?
  - Is for one type of user or for many?
  - What expertise is required?
- How will you make your software available?
- How will your software contribute to research and how will you measure its contribution?

About your software

- What software will you write?
  - What will your software do?
  - Will your software have a name?
- Who are the intended users of your software?
- What software development skills, knowledge and expertise do your users need?
- How will you make your software available to users?
- How will your software contribute to research?

Your software development infrastructure

- What infrastructure will you need, now and in the future?
  - Who needs access?
- Where will your infrastructure be hosted?

Developing good software

- How you will deliver code that can be understood and of good quality?
- How will you choose your test cases?
  - How will you make it easy to write and run tests?
  - How will you ensure that your software is tested regularly?
  - How will you let users know about the tests you do?
  - How will you help developers to understand, modify, extend and test your software?
- Will your software run under multiple environments?
- How will your software and documentation adhere to disability accessibility guidelines?

Managing your dependencies

- What third-party software, models, tools, libraries and services will you use?
- What third-party data sets and online databases will you use?
- What communications protocols and data formats will you use?
- How will you manage and document your dependencies?
- How will you track changes to dependencies?

Managing your software development (1)

- What effort will be available to develop your software?
- How will software development roles be assigned?
- How you will track who is doing what and when it needs to be done by?
- What software development model will you use?

Managing your software development (2)

- How you will manage releases of your software or updates to your services?
- How will you ensure that information is not lost when a developer leaves?
- How often will you review and revise your Software Management Plan?
- How does your Software Management Plan relate to any Data Management Plan?

Engaging with your users

- How will you promote what your software does and who has used it?
- How you will support your users when they ask for help?
- How will your users be able to contribute to your software?

Interested in more?

- Publish a software paper: http://bit.ly/softwarejournals
- Easily archive your GitHub code and make it citable
  - GitHub to Zenodo
  - GitHub to FigShare
- Help Software Citation become widely adopted
  - https://www.force11.org/group/software-citation-implementation-working-group
  - https://doi.org/10.7717/peerj-cs.86
  - https://citeas.org