www.software.ac.uk

Crash Course in Clean Code

https://github.com/softwaresaved/clean-code-workshop

8th December 2017, Open Science for Eco-Evo Research, Amsterdam

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

Who am I?

- Graduated in Physics
- Worked in technology transfer
- Developer and manager on open source scientific software projects
- Started the Software Sustainability Institute based on what I experienced

Who are you?

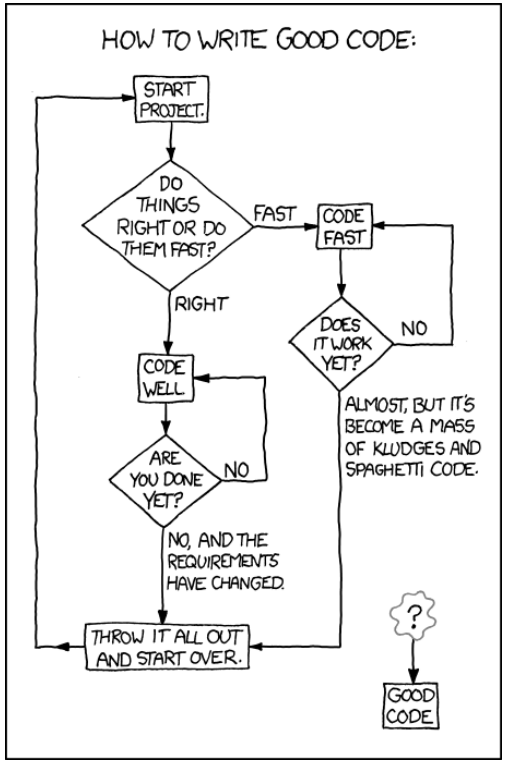
- Your name
- Where you're from
- What you do in one (short) sentence
- One thing you'd like to learn from this workshop

About this workshop

- A “crash course”
  - It won’t teach you everything
  - But it will help you learn what you need to know
- Programming language agnostic
  - But we will be using Python for examples
    - Don’t worry!
- Mixture of discussion and exercises
  - Be prepared to talk to each other!

Identifying good practice

www.software.ac.uk



- It's not always easy to understand how to write good code
- However you probably know more than you think

Identifying good practice

- Split into pairs
- One person in the pair:
  - Suggests practices that improve the process of software development
  - Explains to their partner why this might be useful
- Swap and do the same for the other person
- We'll do this for five minutes each

What's good practice?

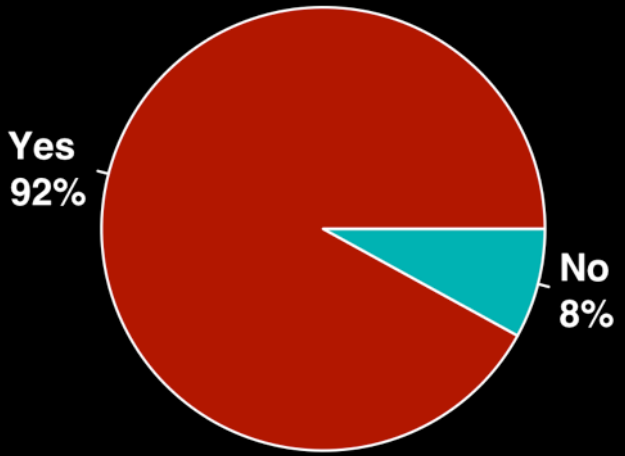
- What things do you think are most important?
- Were there things that were unexpected?
- Are any things which are easier to implement than others?
- Do you think any things are more effective than others?

Why is this important?

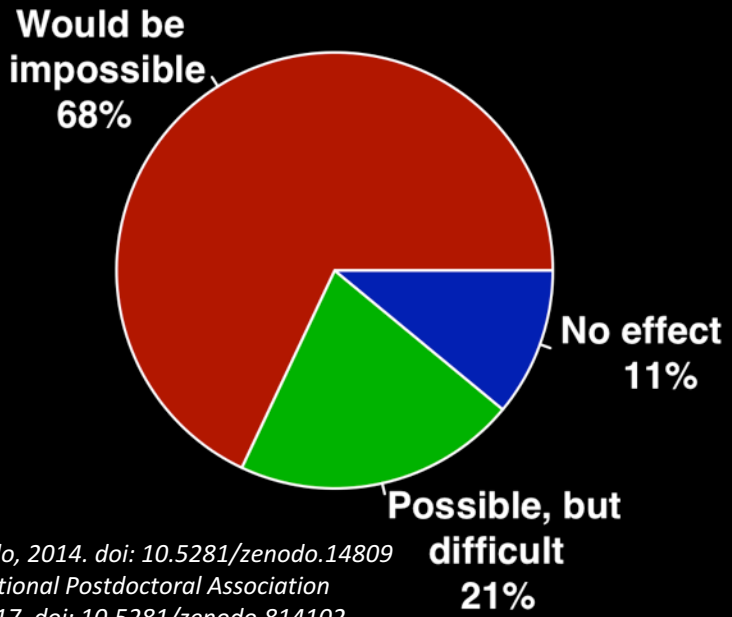
Research depends on software

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

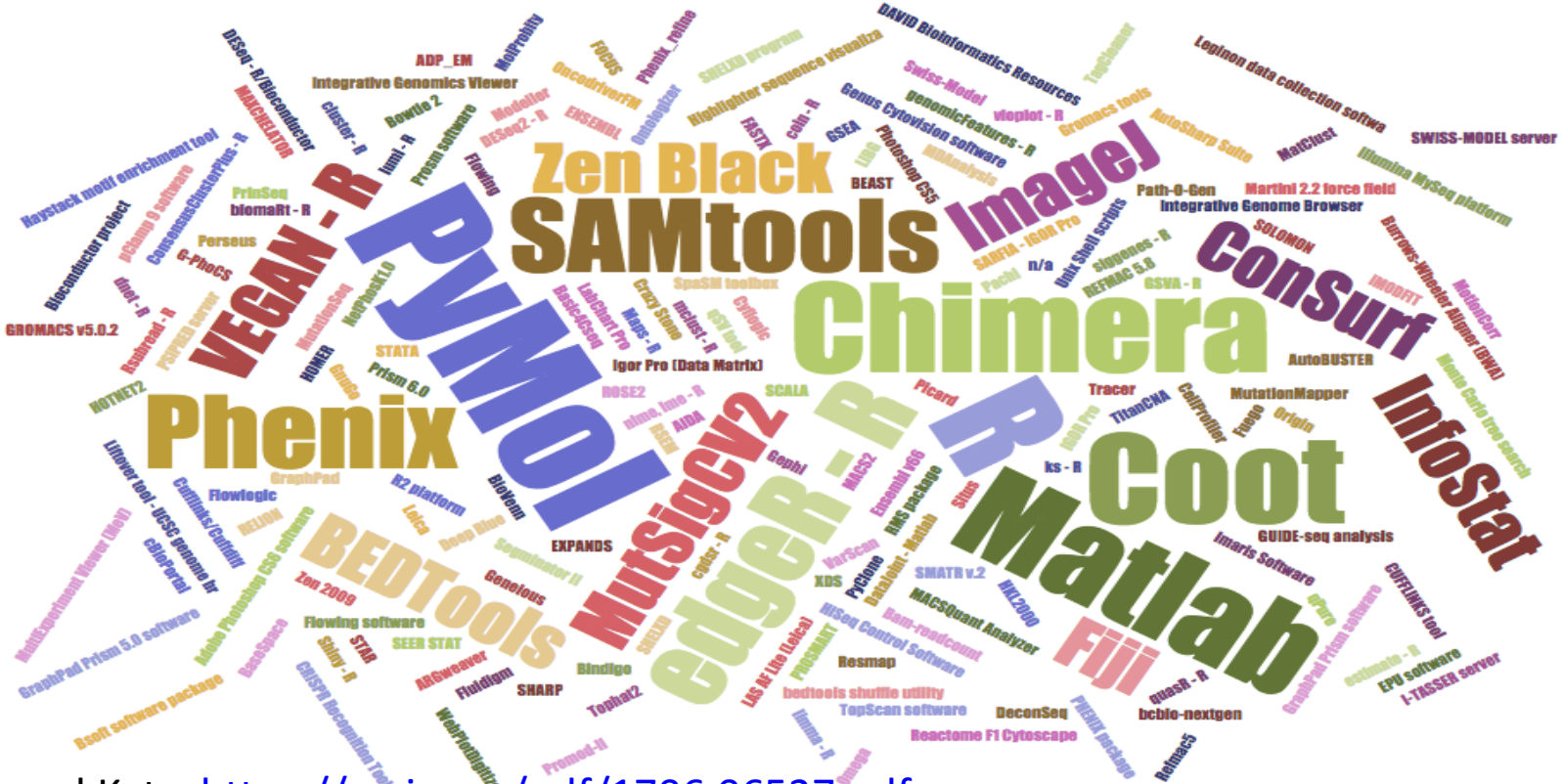
71%

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

“Perceived” importance

- “It has been said … that writing a large piece of software is akin to building infrastructure such as a telescope rather than a creditable scientific contribution…”
- “software development [is] often discounted in the scientific community, and programming is treated as **something to spend as little time on as possible**”
- “Serious scientists are **not expected to carefully test code**, let alone **document** it, in the same way they are trained to properly use other tools or document their experiments”
  - Stodden, V., Bailey, D. H., Borwein, J., LeVeque, R.J., Rider, W., Stein, W. “Setting the Default to Reproducible Reproducibility in Computational and Experimental Mathematics”, ICERM 2013, February 2013.

When things go wrong

www.software.ac.uk

10101010	10101010	10101010	10101010	10101010	10101010	10101010	10101010
10101010	10101010	~~10101010~~	~~10101010~~	~~10101010~~	~~10101010~~	~~10101010~~	~~10101010~~

64 bit floating point number

16 bit signed integer

"nozzle deflections were commanded by the ... software on the basis of data transmitted by the active Inertial Reference System (SRI 2) ... Part of these data ... did not contain proper flight data, but ... was interpreted as flight data."

"SRI 2 did not send correct attitude data ... due to a software exception"

"**software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value**"

ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, 19 July 1996

http://www.esa.int/For_Media/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report

Cost of this bug: US$370 million

http://www.esa.int/spaceinimages/Images/2009/09/Explosion_of_first_Ariane_5_flight_June_4_1996

SCIENTIFIC PUBLISHING

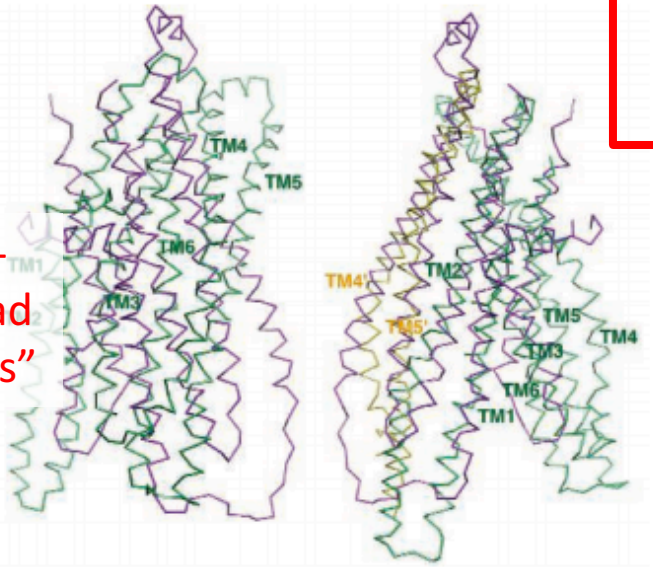
A Scientist's Nightmare: Software Problem Leads to Five Retractions

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, Science researchers published a paper in [illegible] protein structure Chang's group [illegible] 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"

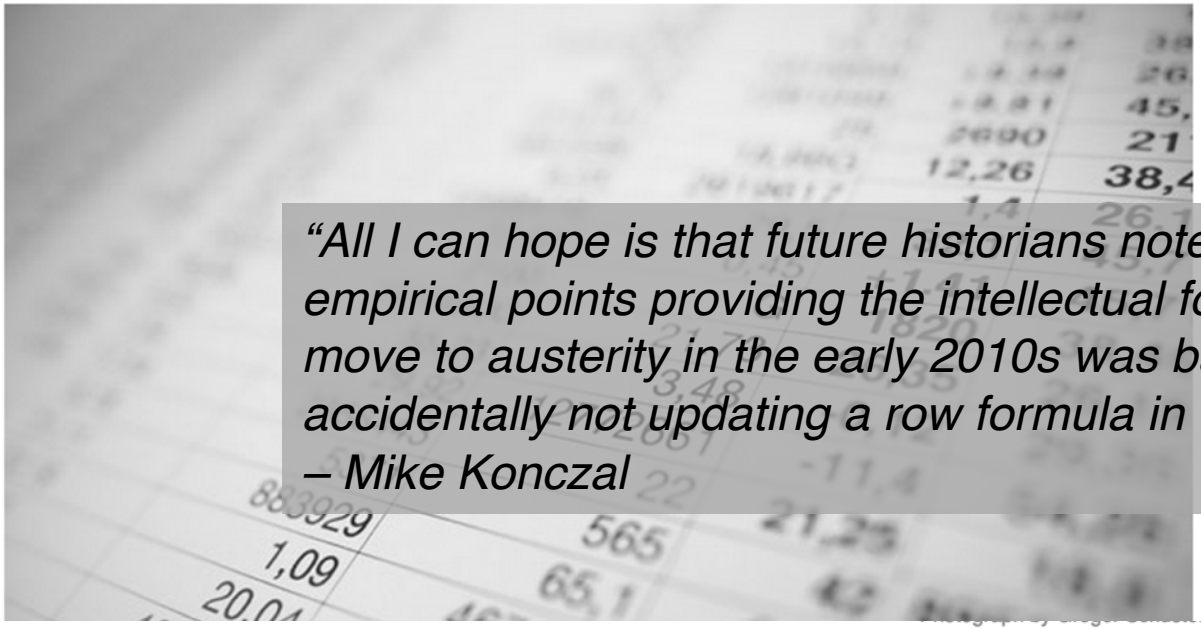
"a homemade data-analysis program had flipped two columns"

www.software.ac.uk

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

FAQ: Reinhart, Rogoff, and the Excel Error That Changed History

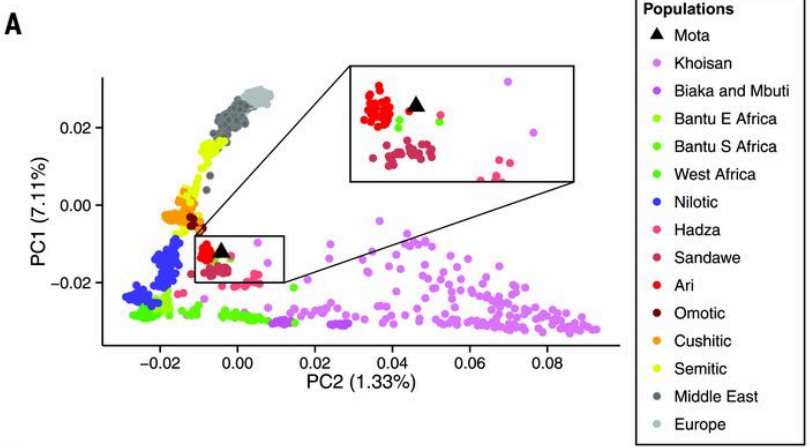
By Peter Coy | April 18, 2013

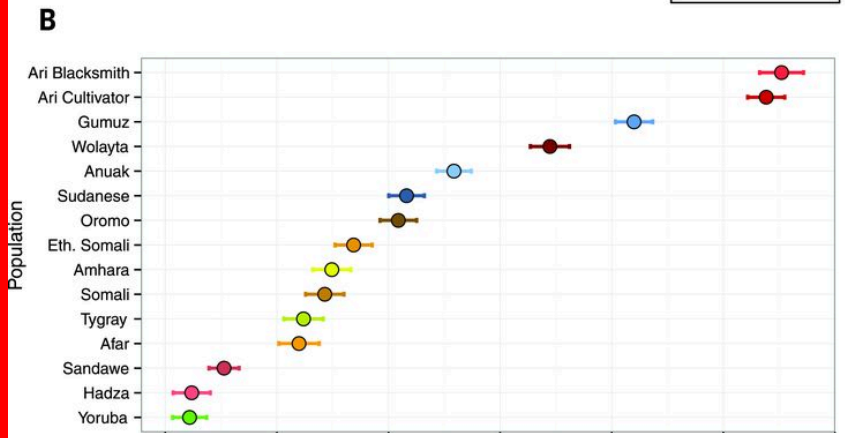


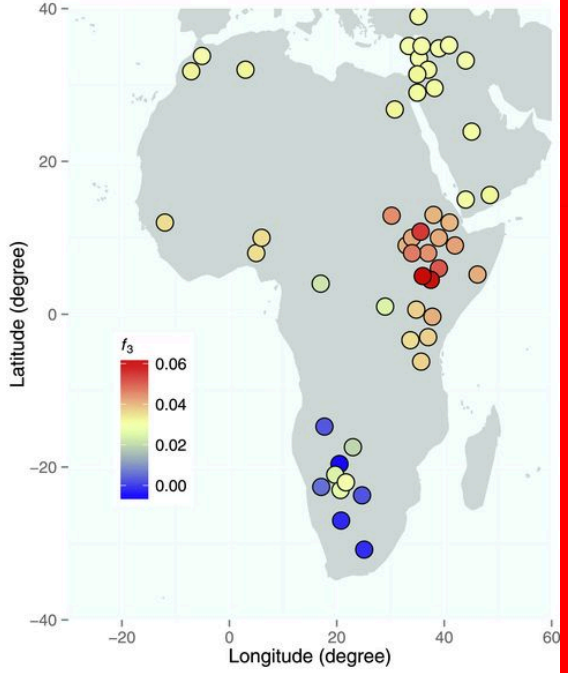
"All I can hope is that future historians note that one of the core empirical points providing the intellectual foundation for the global move to austerity in the early 2010s was based on someone accidentally not updating a row formula in Excel"
– Mike Konczal

Harvard University economists Carmen Reinhart and Kenneth Rogoff have acknowledged making a spreadsheet calculation mistake in a 2010 research paper, "Growth in a Time of Debt" (PDF), which has been widely cited to justify budget-

https://www.bloomberg.com/news/articles/2013-04-18/faq-reinhart-rogoff-and-the-excel-error-that-changed-history







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 https://www.nature.com/news/error-found-in-study-of-first-ancient-african-genome-1.19258

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.

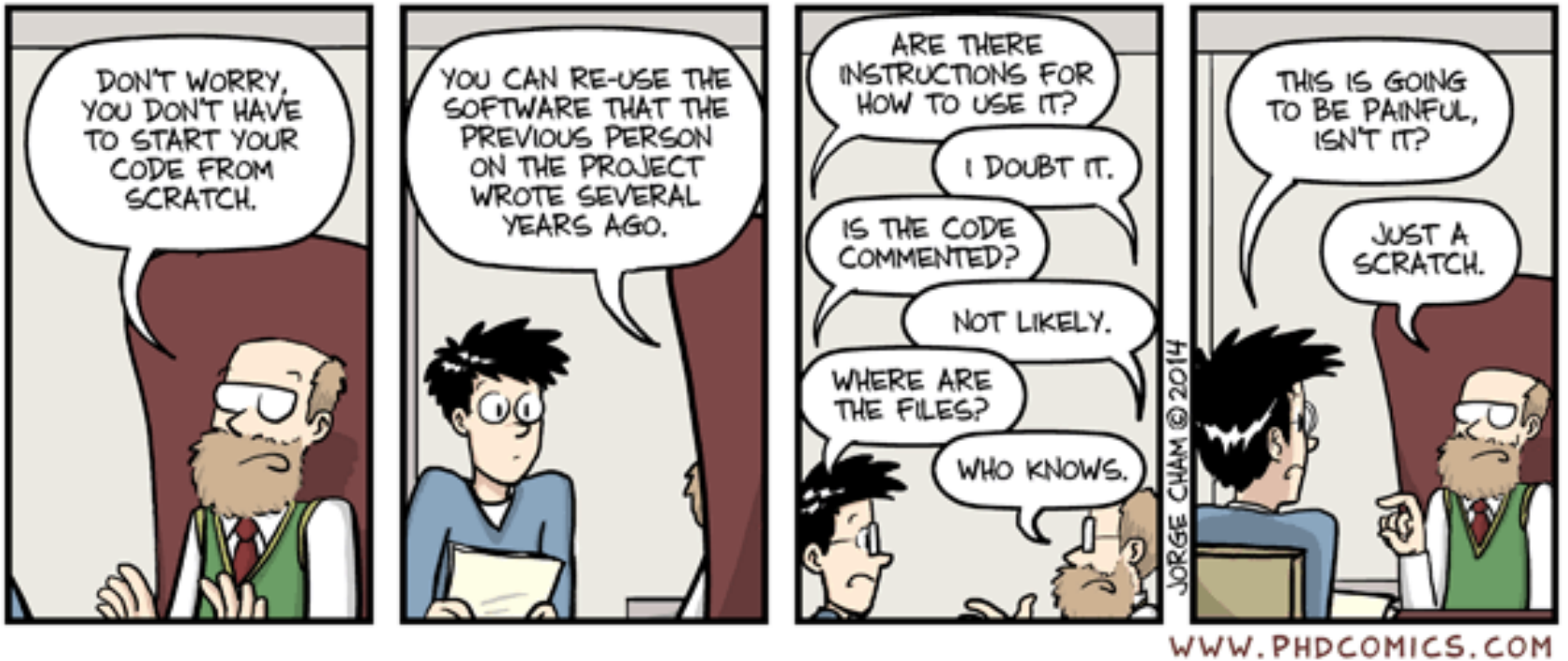
It's still all about reputation

"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

Legacy code

www.software.ac.uk



Piled Higher and Deeper" by Jorge Cham www.phdcomics.com

Everything is legacy code – just wait a few months!

http://phdcomics.com/comics.php?f=1689

Croucher's Law

I can be an idiot and will make mistakes

You are no different!

Corollary: Even when you're an expert programmer

http://mikecroucher.github.io/MLPM_talk/

www.software.ac.uk

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

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?*

Good enough practices to please your future self

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

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

Get some training

www.software.ac.uk

admin@software-carpentry.org

admin@datacarpentry.org

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

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.

What we're trying to avoid

www.software.ac.uk

https://xkcd.com/292/

Code review –
or how we avoid being eaten by raptors

www.software.ac.uk

Most efficient way to find bugs

- “Rigorous inspections can remove up to 90 percent of errors from a software product…”
- “…before the first test case is run”
  - Fact 37, Glass, R. Facts and Fallacies of Software engineering, Addison Wesley, 1992. ISBN-10: 0321117425. ISBN-13: 978-0321117427.
  - Fagan, M.E. “Design and Code inspections to reduce errors in program development”, IBM Systems Journal 15(3), pp182–211, NN 1975. doi:10.1147/sj.153.0182.

Most efficient way to find bugs

- First review and first hour matter most
  - Cohen, J. “Best Kept Secrets of Peer Code Review”, Smart Bear Inc, 2006. ISBN-10: 1599160676. ISBN-13: 978-1599160672.
- Not just for bugs, but alternatives, improvements and two-way knowledge transfer

- Doesn’t have to be “heavyweight” / difficult

Things to look for

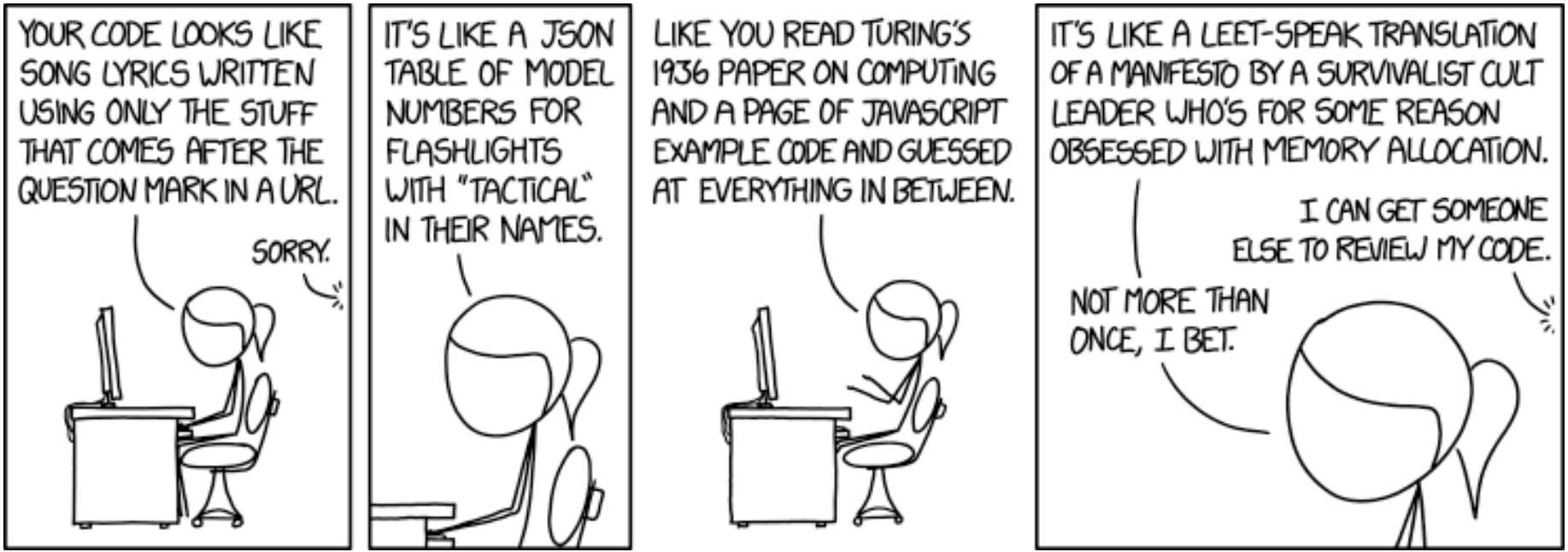
- Is the program broken up into easily understood components?
- Are the components loosely coupled?
- Are names consistent, distinctive and meaningful?
- Is code style and formatting consisten?
- Are constants defined exactly once?
- Does the program have duplicated code?
- Does the program reuse existing libraries?
- Does the documentation describe dependencies and versions?

Things to look for

- Are component interfaces documented?
- Do comments document reasons why code has been implemented as it is, rather than the implementation itself?
- Is documentation embedded in the program?
- Is there user documentation which describes how to use the program?
- Does the program use assertions to check validity of inputs and outputs?
- Does the program have automated tests?

But remember to “play nice”

www.software.ac.uk



From https://xkcd.com/1833/

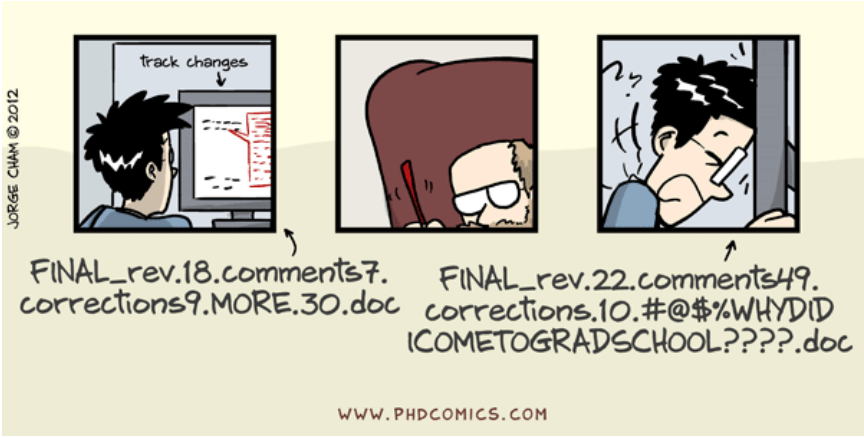
Code review exercise

- In small groups:
  - Take a look at “program.py”
  - Spend 30 minutes reviewing it
  - Write up issues you find
- Then we’ll:
  - Discuss these issues
  - Prioritise them

Learning to love version control

This is version control

www.software.ac.uk

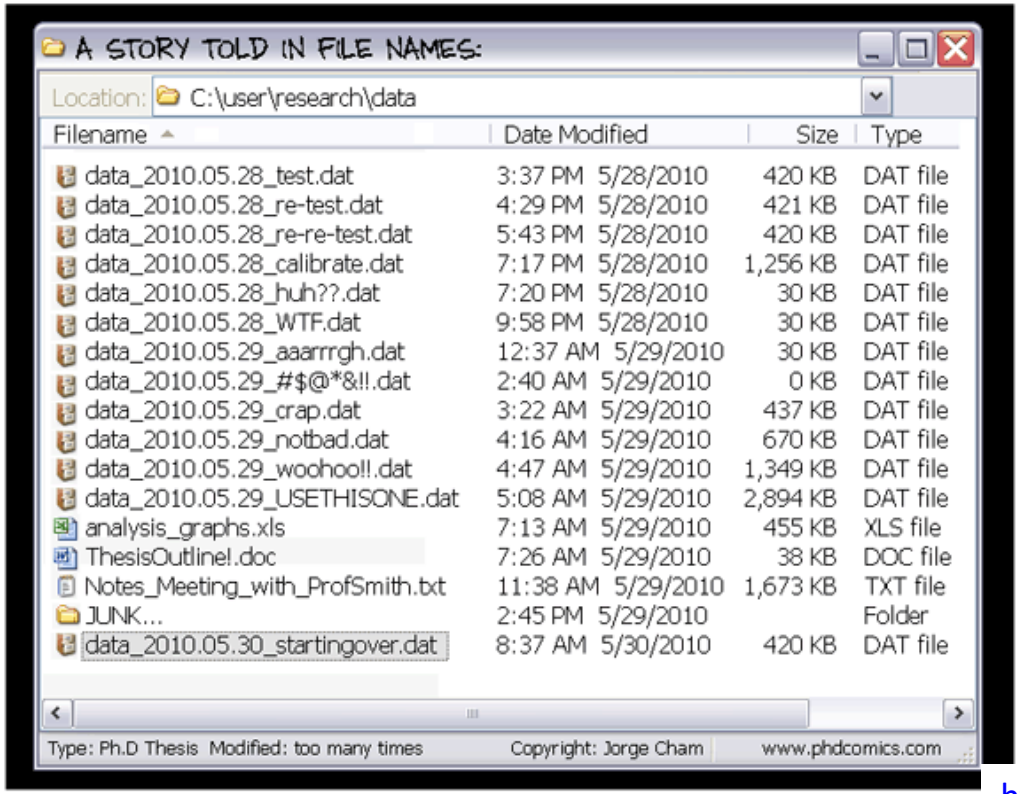


http://phdcomics.com/comics/archive.php?comicid=1531

“Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

This is also version control

www.software.ac.uk



- Everyone's done this at some point in their life
- Are you still doing it now?

Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

http://phdcomics.com/comics/archive.php?comicid=1323

This is not version control

- Change-bound
  - Google Drive: 100 revisions
- Time-bound
  - Google Drive: 30 days
  - DropBox: Last 30 days, 1 year (by subscription)
- What if multiple changes to multiple files need to be treated as a single change?
- How/where to record why changes were made?

Version control / revision control / source code repository

- Unlimited changes
- Unlimited time
- Bound only by available storage
- Treat changes to multiple files as a single change
- Record/view who changed what files, when and why
  - And, for ASCII files, what they changed
- Tag specific versions e.g. “RELEASE-1.0”, “CHEP-2017”
- Retrieve any previous version by version number or tag
- Collaborate on shared files without losing work
- “a lab notebook for code and documents”

Which version to choose?

System

- Client-Server
  - Subversion (SVN)
  - CVS
- Distributed
  - Git
  - Mercurial

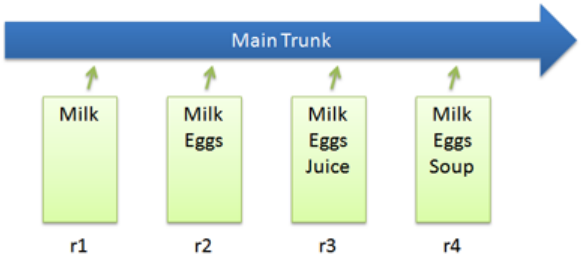
Hosting Platform

- Local
  - SVN
  - Gitlab
- External
  - GitHub
  - BitBucket

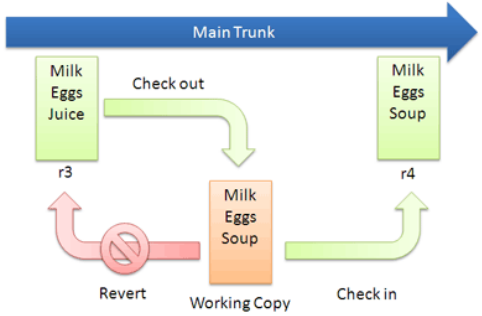
Using version control

www.software.ac.uk

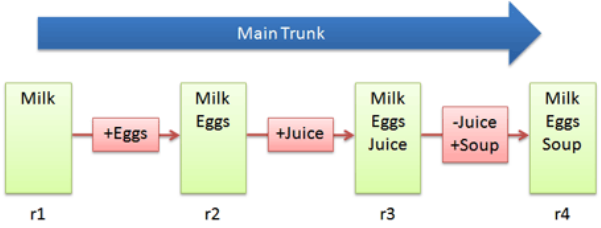
Basic Checkins



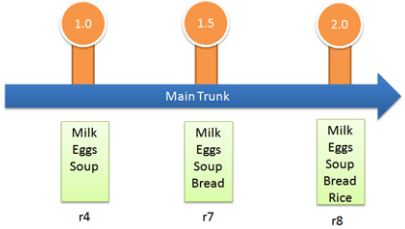
Checkout and Edit



Basic Diffs



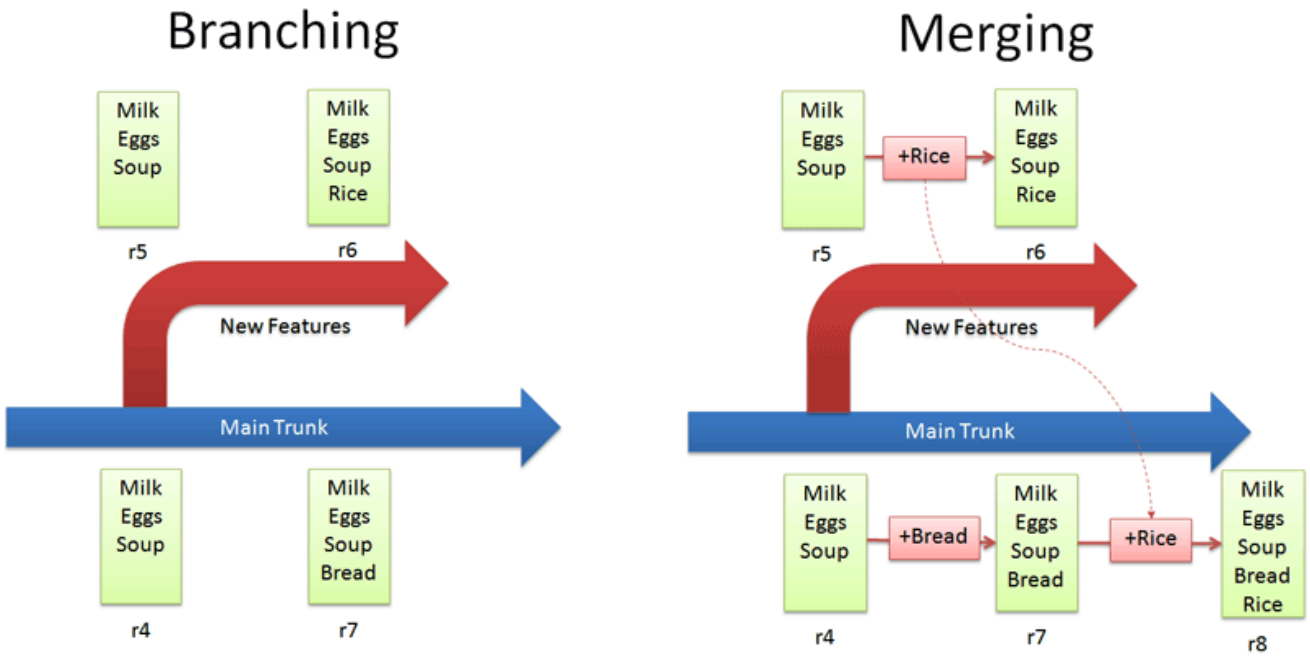
Tagging



https://betterexplained.com/articles/a-visual-guide-to-version-control/

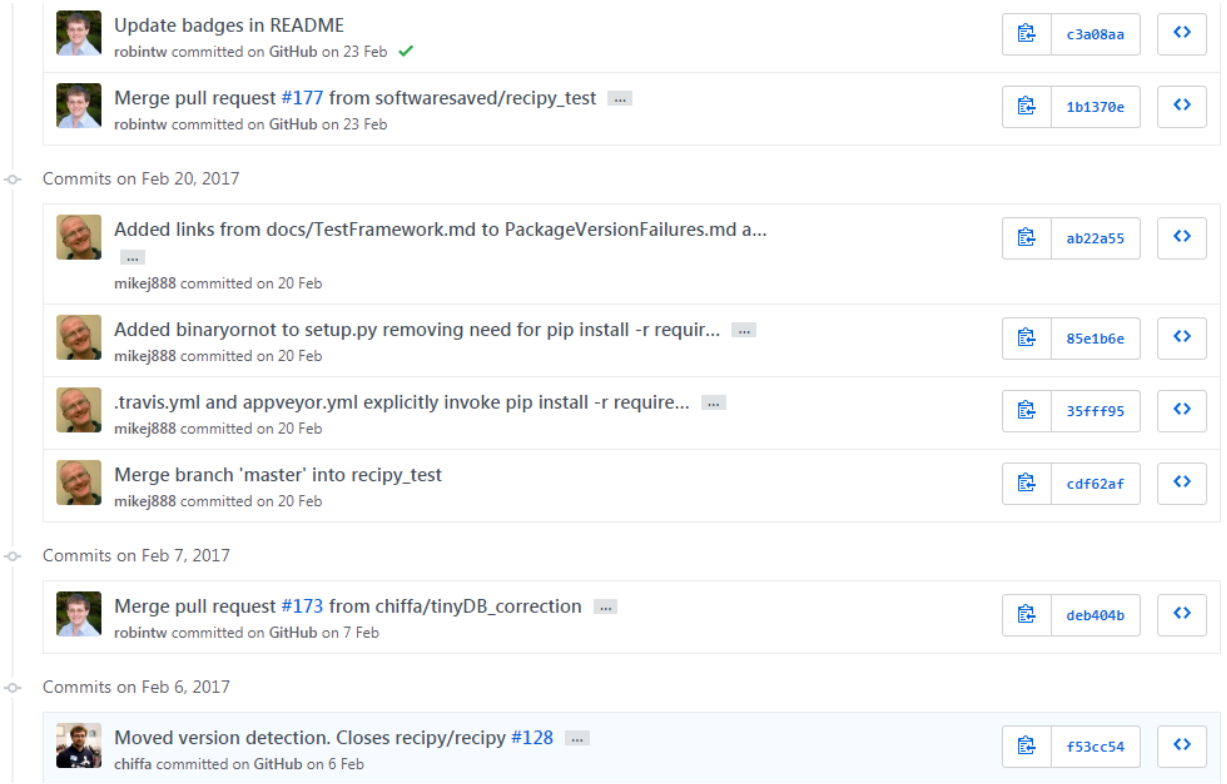
Using version control

www.software.ac.uk



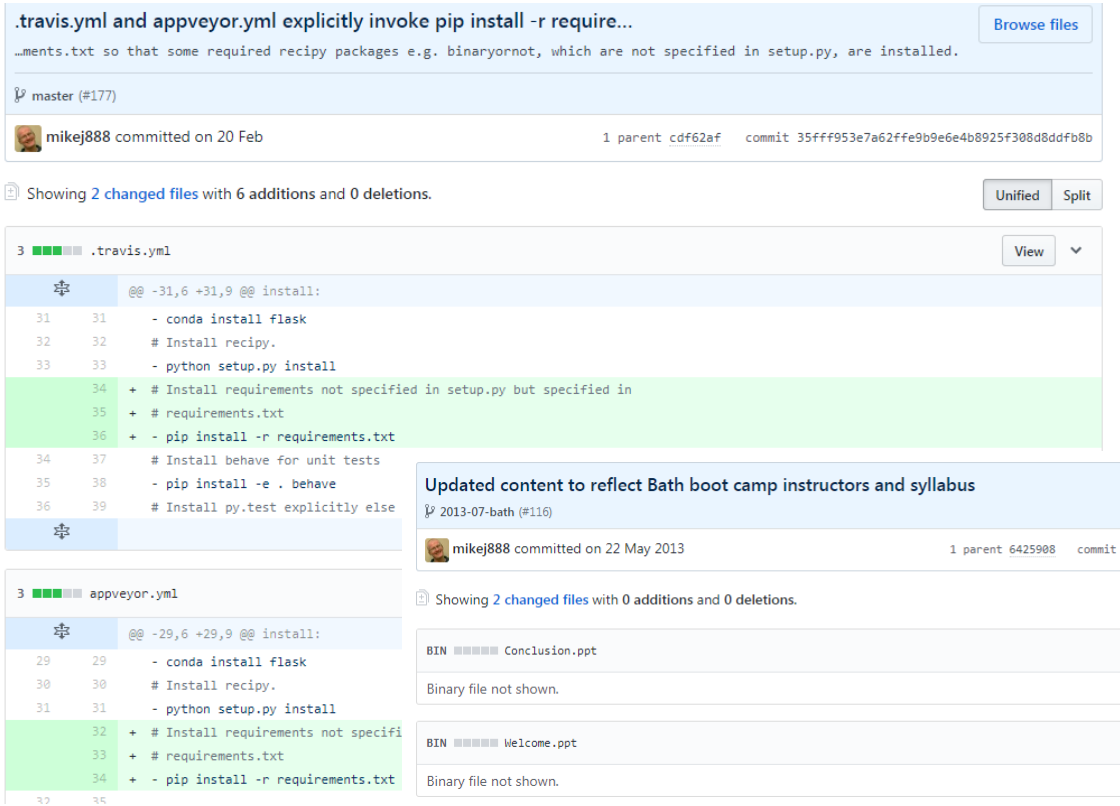
https://betterexplained.com/articles/a-visual-guide-to-version-control/

Record who, what, when & why



Record/view what was changed

www.software.ac.uk



Software Sustainability Institute

Everything created manually

- Source code
  - C, C++, Fortran, Java, Python, R, ...
  - Shell scripts
  - LaTeX
- Word, Powerpoint, Visio
- Build scripts e.g. Makefiles, ANT files
- Configuration files
- Meta-data for images/audio
  - Store images/audio in an archive

The feeling of (version) control

- “Your laptop's just been stolen”
  - “No problem, I’ve pushed regularly to a remote repository”
- “Can I have the code you used to create the data that you graphed in your conference paper?”
  - “Let me check the logs then get the version with that tag”
- “You deleted my analysis section from our paper”
  - “Go and get the previous version from the repository then!”
- “Why did they rewrite my function?”
  - “Look at the commit messages and see why they did it”

Choosing a license

- Don’t overthink it
  - Commercial
    - Retain complete control
  - Open Source – Permissive (MIT, BSD, Apache)
    - Make available as widely as possible
  - Open Source – Copyleft (GPL, LGPL)
    - Ensure that changes are contributed back

Choosing a license

- Think about who you want to use your software and how they will interact with you
- Non-judgemental Guidance
  - https://choosealicense.com/
- Software Licenses in plain English
  - https://tldrlegal.com/

Get an ORCID

www.software.ac.uk

ORCID
Connecting Research and Researchers

EDIT YOUR RECORD | ABOUT ORCID | CONTACT US | HELP

4,143,552 ORCID iDs and counting. See more...

Neil P. Chue Hong

ORCID ID
https://orcid.org/0000-0002-8876-7606

Print view

Other IDs
Scopus Author ID: 8558277300

Biography

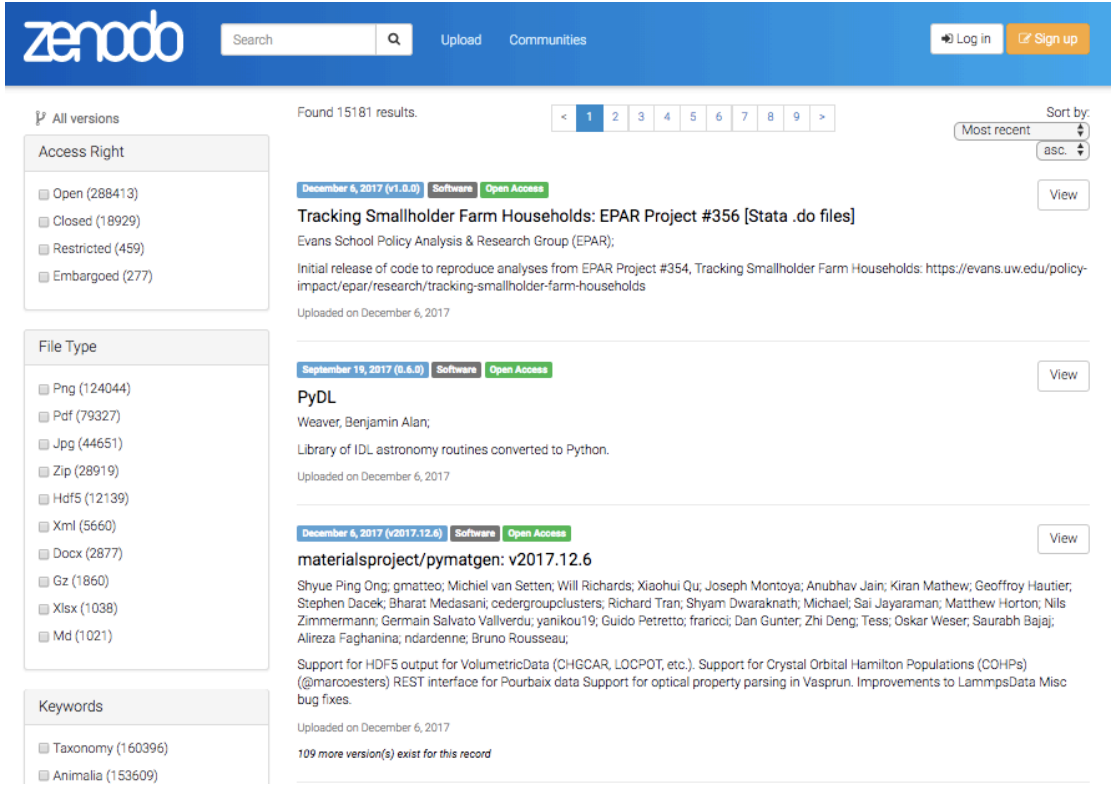
Neil is the founding Director of the Software Sustainability Institute. Graduating with an MPhys in Computational Physics from the University of Edinburgh, he began his career at EPCC, becoming Project Manager there in 2003. During this time he led the Data Access and Integration projects (OGSA-DAI and DAIT), and collaborated in many e-Science projects, including the EU FP6 NextGRID project.
He is presently responsible for representing both the Institute and UK researchers at a national and international level. Within the organisation, he oversees operations, contributes to policy development, develops and manages collaborations, and acts as the principal liaison with stakeholders. His current research is in cloud computing, computational chemistry, software ecosystems, and digital repositories.

Education (1) | Sort

University of Edinburgh: Edinburgh, Edinburgh, United Kingdom
1994-09 to 1999-07 | MPhys (hons) Computational Physics (Physics and Astronomy)

Software in Zenodo

www.software.ac.uk



- Zenodo is a digital repository
- It allows you to deposit software easily

Deposit software in Zenodo

- If your software is in GitHub, it's easy to put in Zenodo:
  - https://guides.github.com/activities/citable-code/
- Gives your code a Digital Object Identifier (DOI)
- Preserves a particular version
  - E.g. used for a paper, major release

Three months from now, you will thank yourself!

www.software.ac.uk

Publish a software paper

- Publish a software paper: http://bit.ly/softwarejournals
  - Methods in Ecology and Evolution
  - Ecography (software notes)
  - Journal of Open Research Software
  - Journal of Open Source Software

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

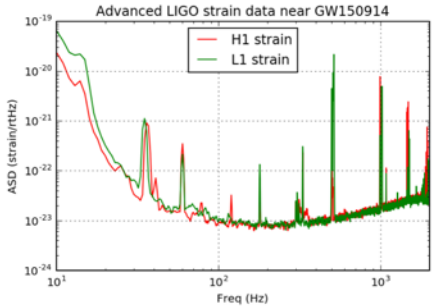
The discovery of the binary pulsar system PSR B1913+16 by Hulse and Taylor [20] and subsequent observations of its energy loss by Taylor and Weisberg [21] demonstrated the existence of gravitational waves. This discovery, along with emerging astrophysical understanding [22], led to the recognition that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

Better program.py

www.software.ac.uk

launch binder

sumofpowers

For a single non-negative integer, calculate the sum of powers that, when added together, give the original integer

In [1]:
```
# sumofpowers
# Author: Neil Chue Hong <N.ChueHong@software.ac.uk>
# Version: 1.1
# Updated: 7 December 2017
# License: BSD-3-Clause (see LICENSE.md)
```

In [2]:
```
#!/usr/bin/env python
import sys
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

The following code block defines the main sumofpowers function, where

- i is the integer
- n is the power

In [14]:
```
def sumofpowers(i, n):
    """Calculate powers of n which sum to i and return list of powers"""
```

Don’t be afraid to publish your code

Research Software Workflow

www.software.ac.uk

describe

GitLab

GitHub

Bitbucket

zenodo

develop → share → preserve

Developed and versioned using code repository

Published via code repository or website

Deposited in digital repository with paper / for preservation

Find out more about the SSI

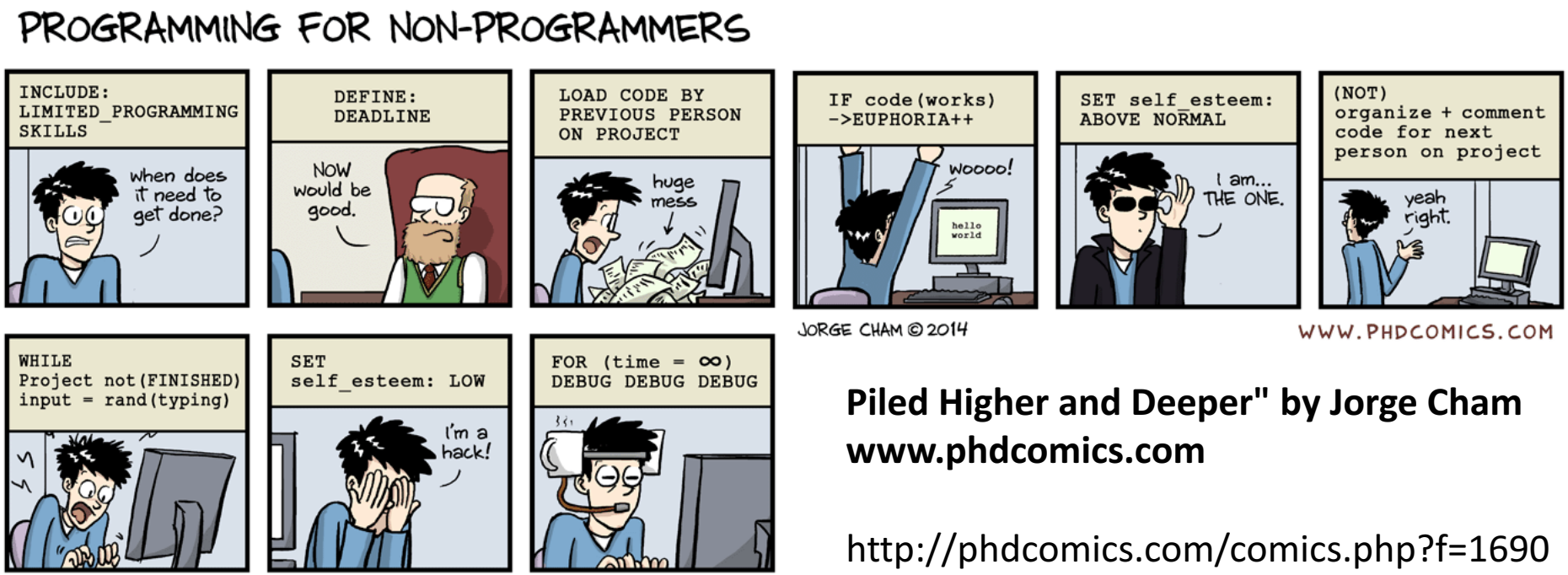
www.software.ac.uk

- 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

This doesn't have to be you!

www.software.ac.uk



Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

http://phdcomics.com/comics.php?f=1690

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

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

Software Sustainability Institute

www.software.ac.uk

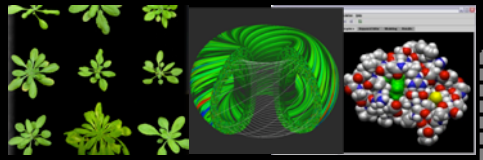
A national facility for cultivating better, more sustainable, research software to enable world-class research

- 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

Software

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



Training

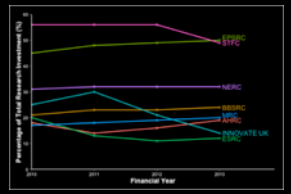
Delivering essential software skills to researchers via CDTs, institutions & doctoral schools

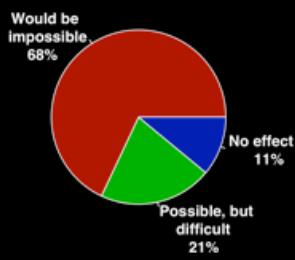
Outreach

Exploiting our platform to enable engagement, delivery & uptake

Policy

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

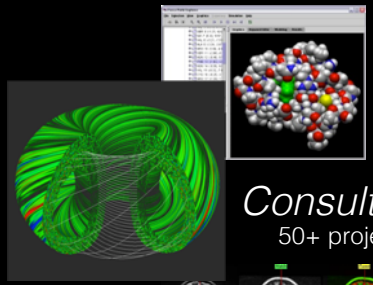




Community

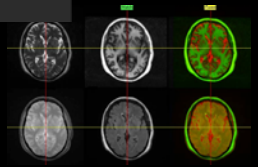
Bringing together the right people to understand and address topical issues

Software



Consultancy

50+ projects



Advice

130+ evaluations
4 surgeries

Training

Courses

35+ UK SWC workshops
1000+ learners

Guides

80+ guides
50,000 readers

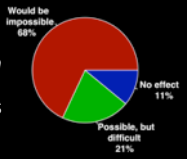
Outreach

Website & blog

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

Research

740 researchers
50,000 grants analysed

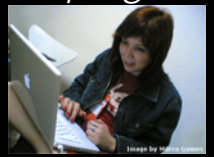


Campaigns

300+ RSEs engaged

BETTER SOFTWARE BETTER RESEARCH

2100 signatures



13 issues highlighted

Policy

Journal of open research software

Workshops

20+ workshops organised

Fellowship

61 domain ambassadors

Community