Paper Discussion

Good enough practices in scientific computing

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLoS computational biology*, *13*(6), e1005510.

Sherine Awad

Repeatability vs Reproducibility

- We say research is repeatable if we can re-run the researchers' experiment using the same method in the same environment and obtain the same results.
- Sharing for repeatability is essential to ensure colleagues and reviewers can evaluate our results based on accurate and complete evidence.
- Sharing for benefaction allows colleagues to build on our results, better advancing scientific progress by avoiding needless replication of work.

- Reproducibility does not necessarily require access to the original research artifacts.
- Rather, it is the independent confirmation of a scientific hypothesis, done post-publication, by collecting different properties from different experiments run on different benchmarks, and using these properties to verify the claims made in the paper.

Collberg, C., & Proebsting, T. A. (2016). Repeatability in computer systems research. *Communications of the ACM*, *59*(3), 62-69.

Problem

Sherine did a ribosome profiling on yeast and wrote up the results for publication. One of the reviewers thinks her broader results are biased by the choice of an aligner and asks her to redo the analysis with a different program

Solution

Sherine repeated her workflow using a different aligner, just changed 3 lines of code to replace bowtie2 with bbmap

Outcome1

- ➔ Sherine discovers that indeed her analysis results are quite different with BBMap.
- ➔ More work is needed.

Outcome2

- ➔ Sherine discovers that her analysis results are very similar whether she uses BBmap or bowtie2 .
- ➔ Her work is done.

Problem

Sherine did a ribosome profiling on yeast and wrote up the results for publication. After publication, another research group led by Angelo (with access to different samples) says that they find a different global pattern of ribosome profiling because both research groups used different samples, data collection approaches, and data analysis approaches, they wonder where the true disagreement lies

Solution

Angelo takes Sherine's pipeline and runs it on their data.

Outcome1

➔ They discover that Angelo's data run with Sherines pipeline gives Sherine's results, and now they can start tracking down what Angelo did differently from Amanda in data analysis.

Outcome2

➔ They discover that Angelo's data run with Sherine's pipeline gives Angelo's results. This suggests that there is something different about the samples or data collection, while the data analysis itself is not the source of the differences.

For science, *reproducibility* is the goal - if other scientists can't follow your process and reach roughly the same results, then the results aren't robust.

If neither Sherine nor Anglelio's data analysis workflow had been easy to repeat, it would have been difficult to know whether data analysis was the source of the differences or not.

- ➔ Repeatability and reproducibility are cornerstones of the scientific process, necessary for avoiding dissemination of flawed results.
- ➔ Repeatability is valuable for efficiency, reuse, and remixing - you and others can repeat, edit, reuse, and repurpose the work.
- ➔ **Repeatability is important for *reproducibility* - if someone else cannot reach approximately the same results with similar input data, then *repeatability* is a requirement for tracking down the differences.**

Collberg, C., & Proebsting, T. A. (2016). Repeatability in computer systems research. *Communications of the ACM*, *59*(3), 62-69.

How to make a computational work repeatable?

Data Management

1. Save the raw data

- ➔ Wherever possible, save the data as originally generated

- ➔ **It is tempting to overwrite raw data files with cleaned-up versions, but faithful retention is essential for rerunning analyses from start to finish, for recovery from analytical mishaps, and for experimenting without fear**

- ➔ Consider changing file permissions to read-only or using spreadsheet protection features so that it is harder to damage raw data by accident or to hand edit it in a moment of weakness.
  Some data will be impractical to manage in this way

- ➔ For example, you should avoid making local copies of large, stable databases

- ➔ **In that case, record the exact procedure used to obtain the raw data, as well as any other pertinent information, such as an official version number or the date of download**

2. Use computer to record all the steps used to process data

If you do not document this step thoroughly, it is impossible for you or any else to repeat the analysis.

Workflow manager

The best way to do this is to write scripts for *every* stage of data processing. This might feel frustratingly slow, but you will get faster with practice

Unique identifiers and verion numbers for

- **Raw data**
- **Programs and libraries**

Values of parameters used to generate any output

Best Practices for Scientific Computing, Aruliah et al, preprint arXIV 2012

BAD SCENARIO

Download some data from Biomart using point/click
Unzip it
Save it as yeastchr1.tsv

Another BAD SCENARIO :

Run a command in the shell:

```
yeastChrI <- getBM(attributes=c('ensembl_gene_id'), filters =
'chromosome_name', values ="I", mart = yeastgenome)
```

3. Use build tools

Instead

Add a script and a rule in your workflow manager

yeastchrI.txt:

```
Rscript getBiomart.R  #This script will download the GO, Chromosomes, GC, and orthogolus data
```

Build tools Tools like Make were originally developed to recompile pieces of software that had fallen out of date. They are now also used to regenerate data and entire papers; when one or more raw input files change, Make can automatically rerun those parts of the analysis that are affected, regenerate tables and plots, and then regenerate the human-readable PDF that depends on them

Dependencies are transitive: if A depends on B and B depends on C, a change to C will indirectly trigger an update to A

Rule 1:

```
yeastchrI.txt :
        Myscript1.py > yeastchrI.txt
```

```
yeastchrI.txt <- getBM(attributes=c('ensembl_gene_id'),
filters = 'chromosome_name', values ="I", mart =
yeastgenome)
```

Rule 2:

```
Myscript2_output.txt: yeastchrI.txt
        Myscript2.py > Myscript2_output.txt
```

make Myscript2_output.txt (will excute Myscript2.py > Myscript2_output.txt)

If rule1 changed:

then make Myscript2_output.txt (will execute rule 1 and rule 2)

DO NOT use mouse to download or to do any step
Please please NO NO NO NO

Every tiny step should be included in your pipeline workflow manager

4. Tools for automated plots

Excel X PLEASE DO NOT

Notebooks like Jupyter
Python
RMarkdown ..etc.

Even tools now take all the pipeline and add latex document with all figures, if any step is changed in the pipeline, figures changes automatically in latex !

5. Use version control

We expect experimental scientist to have lab notebook, version control (i.e github) is the computational version of lab notebook.

script1.final
script1.final.final
script1.final.final.Iswear
Script1.2019.v1 .july20

Even if you number the scripts, which one created the output

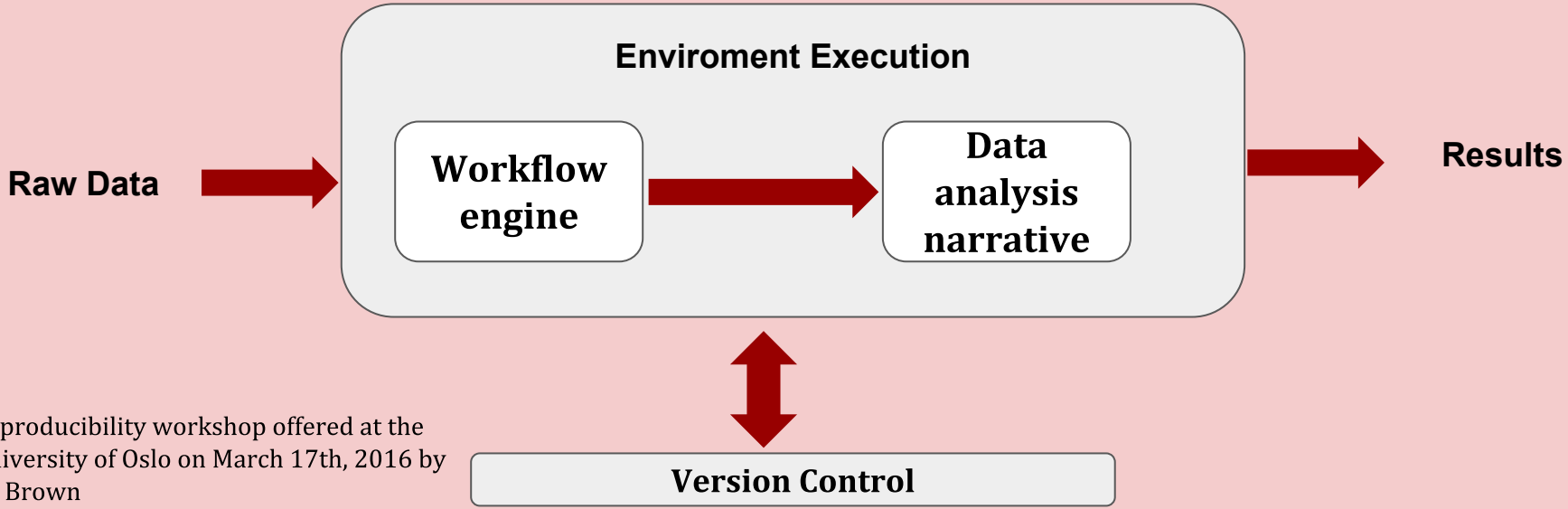
A blunt quote heard in the workshop

If you don't use version control, stop pretending to be a sicentist.

Reproducibility workshop offered at the University of Oslo on March 17th, 2016 by Dr Brown

Workflow engine + **Data analysis narrative** + **Execution Enviroment** + **Versioning**

- Make, Snakemake, Nextflow ..etc
- Jupyter notebook, Python, Rmarkdown ..etc
- Virtual env, Conda, Docker ..etc
- Git ..etc



Reproducibility workshop offered at the University of Oslo on March 17th, 2016 by Dr Brown

The goal of this to have zero chance to say I think I used star 2.x, I assume python2.7 was used, I think it was hg19 version xx.

To make the pipeline repeatable $\Rightarrow$ Zero chance of guess, assumption, ..etc

6. Ensure that raw data are backed up in more than one location

Universities often have their own data-storage solutions

Alternatively, cloud computing resources, like Amazon Simple Storage Service (Amazon S3), Google Cloud Storage,etc.

For large data sets, for which storage and transfer can be expensive and time-consuming, you may need to use incremental backup or specialized storage systems

7. Create the data you wish to see in the world

Convert data from closed, proprietary formats to open, nonproprietary formats that ensure machine readability across time and computing setups. Good options include CSV for tabular data, JSON, YAML, or XML for nontabular data such as graphs (the node-and-arc kind), and HDF5 for certain kinds of structured data

Replace inscrutable variable names and artificial data codes with self-explaining alternatives, e.g., rename variables called name1 and name2 to personal_name and family_name

Recode the treatment variable from 1 vs. 2 to untreated vs. treated
Replace artificial codes for missing data, such as "-99," with NA

For example, a file name like 2016-05-alaska-b.csv makes it easy for both people and programs to select by year or by location

7. Create analysis-friendly data

Analysis can be much easier if you are working with so-called "tidy" data

Two key principles are as follows:
***Make each column a variable*:** Don't cram 2 variables into one; e.g., "male_treated" should be split into separate variables for sex and treatment status. Store units in their own variable or in metadata, e.g., "3.4" instead of "3.4kg"

***Make each row an observation*:** Data often come in a wide format, because that facilitated data entry or human inspection. Imagine 1 row per field site and then columns for measurements made at each of several time points.
Anticipate the need to use multiple tables, and use a unique identifier for every record

8. Submit data to a reputable DOI-issuing repository so that others can access and cite it

Software

1 . Decompose programs into functions

Scroll -scroll -scroll X

Break it into chunks; functions, that’s easier to understand, test, and trouble shoot

2. Don’t repeat yourself or others

2.a) Use lists instead score1, score2, ..etc

2.b) Before writing your own code:

- ➔ **Always search for well-maintained software libraries that do what you need**

- ➔ **Test libraries before relying on them**
  - **DO NOT BLINDLY TRUST SCRIPTS**
  - **PASSING CODE /scripts in emails is a another bad habit**

Painful Scenario

QUAST: quality assessment tool for genome assemblies

A Gurevich, V Saveliev, N Vyahhi, G Tesler - Bioinformatics, 2013 - academic.oup.com

Limitations of genome sequencing techniques have led to dozens of assembly algorithms, none of which is perfect. A number of methods for comparing assemblers have been developed, but none is yet a recognized benchmark. Further, most existing methods for comparing assemblies are only applicable to new assemblies of finished genomes; the problem of evaluating assemblies of previously unsequenced species has not been adequately considered. Here, we present QUAST—a quality assessment tool for evaluating ...

☆ 99 Cited by 2557 Related articles All 18 versions

- ➔ **Issue corrected: people used bugged version coverage function are tracked**
- ➔ **DO NOT BLINDLY TRUST SCRIPTS**
- ➔ **PASSING CODE /scripts in emails is a terrible habit**

2.c Automate repetitive tasks

Workflow managers are needed again

3. Plan for Mistakes:

3.a Defensive Programming

QUAST

An assertion is simply a statement that something holds true at a particular point in a program

```
if cov == 0:
    assert 0, "summed coverage is 0 - this seems like a problem :)"
```

```
assert lines[1].startswith('NUCMER'), lines[0]
```

3.b Write and run tests

The second layer of defense is automated testing.

Automated tests can check to make sure that a single unit of code is returning correct results, or check to make sure that the behavior of a program doesn't change when the details are modified.

These tests are conducted by the computer, so they can be rerun every time the program is modified, to make sure that the changes have not accidentally introduced bugs.

For every bug you find, write a test for it ! Don't say it is fixed.

3.c Optimize software only after it works correctly

The most productive way to make code fast is there to make it work correctly, determine whether it’s actually worth speeding it up, and—in those cases where it is—to use a profiler to identify bottlenecks

3.e Conduct code reviews

Public Communications

Contributers code of conduct

What if the PI is the breacher?

Fix chromosome bug #152

Open **SherineAwad** wants to merge 2 commits into `master` from `fix-chromosome-bug`

Conversation 1 | Commits 2 | Checks 0 | Files changed 1

SherineAwad commented on 27 Feb 2015 Member

Ready For Review

SherineAwad added 2 commits on 27 Feb 2015

fox chromosome bug bc81b7f

Fix bug d3002b2

ctb reviewed on 27 Feb 2015 View changes

scripts/write-snps.py

```
18  -   args=parser.parse_args()
19  -   outfp=args.outfile
14  +   try:
15  +       opts, args = getopt.getopt(argv,"hi:o:",["ofile="])
```

ctb on 27 Feb 2015 Member

why are you using getopt and not argparse?

3.d Document the design and purpose of code rather than its mechanics

Inline documentation that recapitulates code is not useful.

The comment in the code fragment below does nothing to aid comprehension:

i = i + 1 # Increment the variable 'i' by one. UNUSEFUL

Recommend document interfaces and reasons, not implementations.

For example, a clear description at the beginning of a function that describes what it does and its inputs and outputs is useful

Embedded documentation and literate resting

4. Give functions and variables meaningful names

- To document their purpose and to make the program easier to read.
- As a rule of thumb, the greater the scope of a variable, the more informative its name should be;

- While it's acceptable to call the counter variable in a loop i or j, things that are reused often, such as the major data structures in a program, **should *not* have 1-letter names**.

- Remember to follow each language's conventions for names, such as net_charge for Python and NetCharge for Java.

5. Make dependencies and requirements explicit

By adding a file called something like requirements.txt to the root directory of the project

Or by adding a "Getting Started" section to the README file.

6. Do not comment and uncomment sections of code to control a program's behavior

Another terrible habit :)

Makes it difficult or impossible to automate analyses.
Use if/else ..etc

Also, remove commented old code

7. Provide a simple example or test data set

Users (including yourself) can run to determine whether the program is working and **whether it gives a known correct output for a simple known input**

Most of the times when you give an example, you will re-write your documentation :)

8. Submit code to a reputable DOI-issuing repository

Upon submission of paper, just as you do with data.
Your software is as much a product of your research as your papers and should be as easy for people to credit. DOIs for software are provided by Figshare and Zenodo. Zenodo integrates directly with GitHub

Collaborations

1. **Create an overview of your project**
2. **Create a shared "to-do" list**
3. **Make the license explicit**
4. **Make the project citable :** by including a CITATION file in the project's home directory that describes how to cite this project as a whole and where to find (and how to cite) any data sets, code, figures, and other artifacts that have their own DOIs.
5. **5. Decide on communication strategies**

What left out

Are you reproducing the bug?

- ➔ Scientists think of reproducibilty as would other labs get the same results I get?
- ➔ Computational scientists think of reproducibility as are you reproducing the bug?

Unit tests

Cowboy coding (write code! trust results!)

Plan for mistakes! You *will* screw up, and the important thing is not to prevent it in the first place, which is impossible, but to plan around catching them (CTB)

Coverage

Argumentative people are always quick to point out that just because code is executed by a test doesn't mean that it's correct, which is true; my response is that if code isn't executed by *any* test whatsoever, then we *know* it's not being tested. (CTB)

Documentation

Talked about documenting software

We can give a sperate talk later on on nice way of writing documents via github as well.

Continuous integration Automatically run a set of user-defined commands whenever changes are made to a version control repository. These commands typically execute tests to make sure that software hasn't regressed, i.e., that things which used to work still do

Branches A branch is a "parallel universe" within a version control repository. Developers create branches so that they can make multiple changes to a project independently. They are central to the way that experienced developers use systems like Git, but they add an extra layer of complexity to version control for newcomers.

Profiling and performance tuning Profiling is the act of measuring where a program spends its time and is an essential first step in tuning the program (i.e., making it run faster). Both are worth doing but only when the program's performance is actually a bottleneck

A bibliography manager Researchers should use a reference manager of some sort, such as Zotero, and should also obtain and use an ORCID to identify themselves in their publications

Final Comments

- There is a lot more
- These are others' experience, just a guide on how to think and consider reproducibility
- You may with experience add more rules: many rules I added myself as I go
- **Before you "click", think about how this step is gonna be reproduced:** Avoid tools like Ingenuity for repeatability and reproducibility reasons
- **Now people use repeatable pipeline from a to paper automatically**

- Software bugs are inevitable
- Big companies write “Report a bug”
- You will make the case easier to prove its a mistake than a research misconduct
- I am saying that my results are coming from this pipleine, even if the pipeline has a mistake, I am not hiding it

Version control: File issue

NATURE | NEWS

US vaccine researcher sentenced to prison for fraud

The case of Dong-Pyou Han illustrates the uneven nature of penalties for scientific misconduct

Sara Reardon

01 July 2015 | Updated: 01 July 2015

PDF | Rights & Permissions

Charlie Neibergall/AP/PA

Biomedical scientist Dong-Pyou Han (centre) confessed to fabricating and falsifying data on an HIV vaccine.

Rare is the scientist who goes to prison on research misconduct charges. But on 1 July, Dong-Pyou Han, a former biomedical scientist at Iowa State University in Ames, was sentenced to 57 months for fabricating and falsifying data in HIV vaccine trials. Han has also been fined US$7.2

Quotes about Reproducibility

Kohn's Second Law: An experiment is reproducible until another laboratory tries to repeat it

— Alexander Kohn

The natural scientist is concerned with a particular kind of phenomena … he has to confine himself to that which is reproducible … I do not claim that the reproducible by itself is more important than the unique. But I do claim that the unique exceeds the treatment by scientific method. Indeed it is the aim of this method to find and test natural laws…— Wolfgang Pauli

Thank You