Best Practices for Scientific Computing: Principles & MATLAB Examples

Bram Zandbelt
Best Practices for Scientific Computing

Greg Wilson1*, D. A. Aruliah2, C. Titus Brown3, Neil P. Chue Hong4, Matt Davis5, Richard T. Guy6*, Steven H. D. Haddock7, Kathryn D. Huff8, Ian M. Mitchell9, Mark D. Plumbley10, Ben Waugh11*, Ethan P. White12, Paul Wilson13

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 involves 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

Wilson et al. (2014) Best Practices for Scientific Computing. PLoS Biol, 12(1), e1001745.
Best Practices for Scientific Computing

Why is this important?

2014 survey among 417 UK researchers

- **Do you use research software?**
  - Yes: 92%
  - No: 8%

- **What would happen if you could no longer use research software?**
  - It would not be practical to conduct my work without software: 61%
  - It would make no significant difference to my work: 10%
  - My work would require more effort, but it would still be possible: 21%

**We rely heavily on research software**
No software, no research

http://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers
Best Practices for Scientific Computing

https://innoscholemm.silk.co/

Tools that researchers use in practice
Best Practices for Scientific Computing

Why is this important?

**We rely heavily on research software**
No software, no research

**Many of us write software ourselves**
From single-command line scripts to multigigabyte code repositories
adapted from Leek & Peng (2015) *Nature*, 520 (7549), 612.
Best Practices for Scientific Computing

Why is this important?

2014 survey among 417 UK researchers

Do you develop your own research software?

- No 44%
- Yes 56%

We rely heavily on research software
No software, no research

Many of us write software ourselves
From single-command line scripts to multigigabyte code repositories

http://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers
Best Practices for Scientific Computing

Why is this important?

2014 survey among 417 UK researchers

- **We rely heavily on research software**
  No software, no research

- **Many of us write software ourselves**
  From single-command line scripts to multigigabyte code repositories

- **Not all us have had formal training**
  Most of us have learned it “on the street”

http://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers
Best Practices for Scientific Computing

Why is this important?

We rely heavily on research software

No software, no research

Many of us write software ourselves

From single- command line scripts to multigigabyte code repositories

Not all of us have had formal training

Most of us have learned it “on the street”

Poor coding practices can lead to serious problems

Worst case perhaps: 1 coding error led to 5 retractions:
3 in Science, 1 in PNAS, 1 in Molecular Biology

Miller (2006) Science, 314 (5807), 1856-1857.
Best Practices for Scientific Computing

Why now?

Call for increased scrutiny and openness
Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises
Call for increased scrutiny and openness
Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises

Scientific communication relies on evidence that cannot be entirely included in publications, but the rise of computational science has added a new layer of inaccessibility. Although it is now accepted that data should be made available on request, the current regulations regarding the availability of software are inconsistent. We argue that, with some exceptions, anything less than the release of source programs is intolerable for results that depend on computation. The vagaries of hardware, software and natural language will always ensure that exact reproducibility remains uncertain, but withholding code increases the chances that efforts to reproduce results will fail.

Ince et al (2012) Nature, 482 (7386) 485-488.
Best Practices for Scientific Computing

Why now?

Call for increased scrutiny and openness

Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises

Peng (2011) Science, 334 (6060) 1226-1227
Best Practices for Scientific Computing

Why now?

Call for increased scrutiny and openness
Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises

Nosek et al. (2015) Science, 348 (6242) 1422-1425.
**Best Practices for Scientific Computing**

Why now?

---

**Call for increased scrutiny and openness**

Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises

---

Nosek et al. (2015) Science, 348 (6242) 1422-1425.
Best Practices for Scientific Computing

Why now?

Call for increased scrutiny and openness
Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises
Best Practices for Scientific Computing

Why now?

Call for increased scrutiny and openness
Following cases of scientific misconduct, widespread questionable research practices, and reproducibility crises

http://www.nwo.nl/binaries/content/assets/nwo/documents/nwo/open-science/open-access-flyer-2012-def-eng-scherm.pdf
http://ec.europa.eu/research/participants/data/ref/h2020/grants_manual/hi/oa_pilot/h2020-hi-oa-pilot-guide_en.pdf
Best Practices for Scientific Computing

What should we (try to) do?

1. **Write programs for people**
   so you and others quickly understand the code
2. **Let the computer do the work**
   to increase speed and reproducibility
3. **Make incremental changes**
   to increase productivity and reproducibility
4. **Don’t repeat yourself (or others)**
   repetitions are vulnerable to errors and a waste of time
5. **Plan for mistakes**
   almost all code has bugs, but not all bugs cause errors
6. **Optimize software only after it works**
   and perhaps only if it is worth the effort
7. **Document design and purpose, not implementation**
   to clarify understanding and usage
8. **Collaborate**
   to maximize reproducibility, clarity, and learning
Best Practices for Scientific Computing

1. Write programs for people, not computers
   … so you and others quickly understand the code

   A program should not require its readers to hold more than a handful of facts in memory at once
   Break up programs in easily understood functions, each of which performs an easily understood task
1. Write programs for people, not computers
   … so you and others quickly understand the code
Best Practices for Scientific Computing

1. Write programs for people, not computers
   … so you and others quickly understand the code

   A program should not require its readers to hold more than a handful of facts in memory at once
   Break up programs in easily understood functions, each of which performs an easily understood task

   Make names consistent, distinctive, and meaningful
   e.g. fileName, meanRT rather than, tmp, result
   e.g. camelCase, pothole_case, UPPERCASE
A program should not require its readers to hold more than a handful of facts in memory at once. Break up programs in easily understood functions, each of which performs an easily understood task.

Make names consistent, distinctive, and meaningful:
- e.g. `fileName`, `meanRT` rather than, `tmp`, `result`
- e.g. `camelCase`, `pothole_case`, `UPPERCASE`

Make code style and formatting consistent:
- Use existing guidelines and check lists.

CODING CHECKLIST:

**GENERAL**
- Do you have a good design?
- Is the code traceable to requirements?
- Does the code support automated use and testing?
- Does the code have documented test cases?
- Is the code adheres to your designated coding style and standard?
- Is the code free from Code Analyzer messages?
- Does each function perform one well-defined task?
- Is each class cohesive?
- Have you refactored any blocks of code repeated unnecessarily?
- Are there unnecessary global constants or variables?
- Does the code have appropriate error checking and handling?
- If performance is an issue, have you profiled the code?

**UNDERSTANDABILITY**
- Is the code straightforward and does it avoid "cleverness"?
- Do you thoroughly understand your code? Will anyone else?
- Is it easy to follow and understand?

**CODE CHANGES**
- Have the changes been reviewed as thoroughly as initial development would be?
- Do the changes enhance the program's internal quality rather than degrading it?
- Have you improved the system's modularity by breaking functions into smaller functions?
- Have you introduced any flaws or limitations?
- rj@datatool.com  www.datatool.com

http://nl.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0
http://nl.mathworks.com/matlabcentral/fileexchange/48121-matlab-coding-checklist
Best Practices for Scientific Computing

2. Let the computer do the work
   … to increase speed and reproducibility

   Make the computer repeat tasks
   Go beyond the point-and-click approach
Best Practices for Scientific Computing

2. Let the computer do the work
   … to increase speed and reproducibility

   Make the computer repeat tasks
   Go beyond the point-and-click approach

   Save recent commands in a file for re-use
   Log session’s activity to file
   In MATLAB: diary
Best Practices for Scientific Computing

2. Let the computer do the work
   … to increase speed and reproducibility

Make the computer repeat tasks
Go beyond the point-and-click approach

Save recent commands in a file for re-use
Log session’s activity to file
In MATLAB: diary

Use a build tool to automate workflows
e.g. SPM’s matlabbatch, IPython/Jupyter notebook, SPSS syntax

*SPM’s matlabbatch*
Best Practices for Scientific Computing

2. Let the computer do the work

http://jupyter.org/
3. Make incremental changes
   … to increase productivity and reproducibility

   Work in small steps with frequent feedback and course correction
   Agile development: add/change one thing at a time, aim for working code at completion
Best Practices for Scientific Computing

3. Make incremental changes
   … to increase productivity and reproducibility

   Work in small steps with frequent feedback and course correction
   Agile development: add/change one thing at a time, aim for working code at completion

   Use a version control system
   Documents who made what contributions when
   Provides access to entire history of code + metadata
   Many free options to choose from: e.g. git, svn

http://www.phdcomics.com/comics/archive.php?comicid=1531
Best Practices for Scientific Computing

Changes are saved sequentially ("snapshots")

Different versions can be saved

Multiple versions can be merged

http://swcarpentry.github.io/git-novice/01-basics.html
Best Practices for Scientific Computing

3. Make incremental changes
   … to increase productivity and reproducibility

Work in small steps with frequent feedback and course correction
Agile development: add/change one thing at a time, aim for working code at completion

Use a version control system
Documents who made what contributions when
Provides access to entire history of code + metadata
Many free options to choose from: e.g. git, svn

Put everything that has been created manually in version control
Advantages of version control also apply to manuscripts, figures, presentations

http://www.phdcomics.com/comics/archive.php?comicid=1531
Best Practices for Scientific Computing

https://github.com/smathot/materials_for_P0001/commits/master/manuscript
Best Practices for Scientific Computing

4. Don’t repeat yourself (or others)
   … repetitions are vulnerable to errors and a waste of time

Every piece of data must have a single authoritative representation in the system
Raw data should have a single canonical version

http://swcarpentry.github.io/v4/data/mgmt.pdf
4. Don’t repeat yourself (or others) … repetitions are vulnerable to errors and a waste of time

http://swcarpentry.github.io/v4/data/mgmt.pdf
4. Don’t repeat yourself (or others)
… repetitions are vulnerable to errors and a waste of time

Every piece of data must have a single authoritative representation in the system
Raw data should have a single canonical version

Modularize code rather than copying and pasting
Avoid “code clones”, they make it difficult to change code as you have to find and update multiple fragments
4. Don’t repeat yourself (or others) … repetitions are vulnerable to errors and a waste of time

code clone

SCRIPT X
Statements
....
Statements
....

SCRIPT Y
Statements
....
Statements
....

SCRIPT X
Call function a
Call function b

SCRIPT Y
Call function a
Call function c
4. Don’t repeat yourself (or others)
… repetitions are vulnerable to errors and a waste of time

Every piece of data must have a single authoritative representation in the system
Raw data should have a single canonical version

Modularize code rather than copying and pasting
Avoid “code clones”, they make it difficult to change code as you have to find and update multiple fragments

Re-use code instead of rewriting it
Chances are somebody else made it: e.g. look at SPM Extensions (~200), NITRC (810), CRAN (7395), MATLAB File Exchange (25297), PyPI (68247)
5. Plan for mistakes

… almost all code has bugs, but not all bugs cause errors

**Add assertions to programs to check their operation**

Assertions are internal self-checks that verify whether inputs, outputs, etc. are line with the programmer’s expectations

In MATLAB: `assert`
5. Plan for mistakes

... almost all code has bugs, but not all bugs cause errors

**Add assertions to programs to check their operation**

Assertions are internal self-checks that verify whether inputs, outputs, etc. are line with the programmer’s expectations

In MATLAB: `assert`

**Use an off-the-shelf unit testing library**

Unit tests check whether code returns correct results

In MATLAB: `runtests`
5. Plan for mistakes

… almost all code has bugs, but not all bugs cause errors

Add assertions to programs to check their operation
Assertions are internal self-checks that verify whether inputs, outputs, etc. are line with the programmer’s expectations
In MATLAB: assert

Use an off-the-shelf unit testing library
Unit tests check whether code returns correct results
In MATLAB: runtests

Turn bugs into test cases
To ensure that they will not happen again
5. Plan for mistakes

… almost all code has bugs, but not all bugs cause errors

Add assertions to programs to check their operation
Assertions are internal self-checks that verify whether inputs, outputs, etc. are line with the programmer’s expectations
In MATLAB: `assert`

Use an off-the-shelf unit testing library
Unit tests check whether code returns correct results
In MATLAB: `runtests`

Turn bugs into test cases
To ensure that they will not happen again

Use a symbolic debugger
Symbolic debuggers make it easy to improve code
Can detect bugs and sub-optimal code, run code step-by-step, track values of variables
Best Practices for Scientific Computing

```matlab
function F = exgauss_cdf(y,X)
% DESCRIPTION
% Computes cumulative distribution function for each of the data points in y, given the vector of ex-Gaussian parameters X
% SYNTAX
% F = EXGAUSS_CDF(y,X);
% EXAMPLES
% Mu = 400;
% Sigma = 50;
% Tau = 100;
% X = [Mu, Sigma, Tau];
% y = 200:0.1:1800;
% F = EXGAUSS_CDF(y,X);

Mu = X(1);
Sigma = X(2);
Tau = X(3);

% Compute cumulative distribution of ex-Gaussian
part1 = -exp(-y./Tau + Mu/Tau + Sigma.^2./2./Tau.^2);
part2 = normcdf(y-Mu-Sigma.^2./2./Tau,.5);
part3 = normcdf(y-Mu)/Sigma;
F = part1.*part2 + part3;
```

REFERENCES

% Bram Zandbelt, bramzandbelt@gmail.com
% Created: Fri 24 Jan 2014 16:18:10 CST by bram
% Modified: Fri 24 Jan 2014 16:18:10 CST by bram

% Decode parameters
Mu = X(1);
Sigma = X(2);
Tau = X(3);

% Compute cumulative distribution of ex-Gaussian
part1 = -exp(-y./Tau + Mu/Tau + Sigma.^2./2./Tau.^2);
part2 = normcdf(y-Mu-Sigma.^2./2./Tau,.5);
part3 = normcdf(y-Mu)/Sigma;
F = part1.*part2 + part3;
```
Best Practices for Scientific Computing
Best Practices for Scientific Computing
6. Optimize software only after it works correctly 
   … and perhaps only if it is worth the effort

Use a profiler to identify bottlenecks
Useful if your code takes a long time to run
In MATLAB: `profile`
Best Practices for Scientific Computing

6. Optimize software only after it works correctly
   … and perhaps only if it is worth the effort

Use a profiler to identify bottlenecks
Useful if your code takes a long time to run
In MATLAB: `profile`

Write code in the highest-level language possible
Programmers produce roughly same number of lines of code per day, regardless of language
High-level languages achieve complex computations in few lines of code
7. Document design and purpose, not mechanics

… good documentation helps people understand code

Document interfaces and reasons, not implementations
Write structured and informative function headers
Write a README file for each set of functions

see: http://www.mathworks.com/matlabcentral/fileexchange/4908-m-file-header-template/content/template_header.m
7. Document design and purpose, not mechanics
   … good documentation helps people understand code

**Document interfaces and reasons, not implementations**
Write structured and informative function headers
Write a README file for each set of functions

**Refactor code in preference to explaining how it works**
Restructuring provides clarity, reducing space/time needed for explanation
7. Document design and purpose, not mechanics
   … good documentation helps people understand code

**Document interfaces and reasons, not implementations**
Write structured and informative function headers
Write a README file for each set of functions

**Refactor code in preference to explaining how it works**
Restructuring provides clarity, reducing space/time needed for explanation

**Embed the documentation for a piece of software in that software**
Embed in function headers, README files, but also interactive notebooks (e.g. knitr, IPython/Jupyter)
Increases likelihood that code changes will be reflected in the documentation
8. Collaborate

Use (pre-merge) code reviews
Like proof-reading manuscripts, but then for code
Aims to identify bugs, improve clarity, but other benefits are spreading knowledge, learning to coded
8. Collaborate

Use (pre-merge) code reviews
Like proof-reading manuscripts, but then for code
Aims to identify bugs, improve clarity, but other benefits
are spreading knowledge, learning to coded

Use pair programming when bringing someone new
up to speed and when tackling tricky problems
Intimidating, but really helpful
8. Collaborate

Use (pre-merge) code reviews
Like proof-reading manuscripts, but then for code
Aims to identify bugs, improve clarity, but other benefits
are spreading knowledge, learning to coded

Use pair programming when bringing someone new
up to speed and when tackling tricky problems
Intimidating, but really helpful

Use an issue tracking tool
Perhaps mainly useful for large, collaborative coding
efforts, but many code sharing platforms (e.g. Github,
Bitbucket) come with these tools
Best Practices for Scientific Computing

Useful resources
