OPEN ACCESS Freely available online

PLOS | COMPUTATIONAL BIOLOGY

Editorial

Ten Simple Rules for Effective Computational Research

James M. Osborne[1,2]*, Miguel O. Bernabeu[3,4], Maria Bruna[1,2], Ben Calderhead[3], Jonathan Cooper[1], Neil Dalchau[2], Sara-Jane Dunn[2], Alexander G. Fletcher[5], Robin Freeman[2,3], Derek Groen[4], Bernhard Knapp[6], Greg J. McInerny[1,2], Gary R. Mirams[1], Joe Pitt-Francis[1], Biswa Sengupta[7], David W. Wright[3,4], Christian A. Yates[5], David J. Gavaghan[1], Stephen Emmott[2], Charlotte Deane[6]

1 Computational Biology Group, Department of Computer Science, University of Oxford, Wolfson Building, Oxford, United Kingdom, **2** Computational Science Laboratory, Microsoft Research, Cambridge, United Kingdom, **3** CoMPLEX, Mathematical and Physical Sciences, University College London, Physics Building, London, United Kingdom, **4** Centre for Computational Science, Department of Chemistry, University College London, London, United Kingdom, **5** Wolfson Centre for Mathematical Biology, Mathematical Institute, University of Oxford, Andrew Wiles Building, Radcliffe Observatory Quarter, Oxford, United Kingdom, **6** Department of Statistics, University of Oxford, Oxford, United Kingdom, **7** The Wellcome Trust Centre for Neuroimaging, Institute of Neurology, University College London, London, United Kingdom

In order to attempt to understand the complexity inherent in nature, mathematical, statistical and computational techniques are increasingly being employed in the life sciences. In particular, the use and development of software tools is becoming vital for investigating scientific hypotheses, and a wide range of scientists are finding software development playing a more central role in their day-to-day research. In fields such as biology and ecology, there has been a noticeable trend towards the use of quantitative methods for both making sense of ever-increasing amounts of data [1] and building or selecting models [2].

As Research Fellows of the "2020 Science" project (http://www.2020science.net), funded jointly by the EPSRC (Engineering and Physical Sciences Research Council) and Microsoft Research, we have firsthand experience of the challenges associated with carrying out multidisciplinary computation-based science [3–5]. In this paper we offer a jargon-free guide to best practice when developing and using software for scientific research. While many guides to software development exist, they are often aimed at computer scientists [6] or concentrate on large open-source projects [7]; the present guide is aimed specifically at the vast majority of scientific researchers: those without formal training in computer science. We present our ten simple rules with the aim of enabling scientists to be more effective in undertaking research and therefore maximise the impact of this research within the scientific community. While these rules are described individually, collectively they form a single vision for how to approach the practical side of computational science.

Our rules are presented in roughly the chronological order in which they should be undertaken, beginning with things that, as a computational scientist, you should do *before* you even think about writing any code. For each rule, guides on getting started, links to relevant tutorials, and further reading are provided in the supplementary material (Text S1).

Rule 1: Look Before You Leap

One of the key considerations in the development of any method, computational or otherwise, is whether it has previously been approached by someone else. A growing wealth of software toolboxes and libraries exist to tackle many problems. However, assessing the range and quality of what is available can be hard, especially when addressing nontraditional problems. A simple but often-overlooked approach is to conduct a software literature review to ascertain what software is available and has been successfully employed. Software repositories (e.g., GitHub, https://github.com/, and SourceForge, http://sourceforge.net/) are a good place to begin a review. Furthermore, engaging with the network of researchers surrounding your own is invaluable; see [8] and [9] for advice on this. If your coworkers write software in the same language or use particular toolboxes, you may be able to consult their expertise in order to accelerate and provide support for your work.

Rule 2: Develop a Prototype First

Before writing any code, it is imperative to clarify what you are trying to implement: what functionality do you require, and what interfaces do you need? When implementing your latest developments, you should first begin by considering a prototype (i.e., a simplified version of the full system or algorithm) to gain insight and to guide the next steps. This is equally relevant whether building on existing code or starting from scratch. By prototyping new functionality and building code up incrementally, you can check that each element of your code operates as expected (and each incremental development can be tested; see Rule 8). Breaking your problem up into smaller elements like this will also help to provide structure to your code and will make it much easier when you subsequently need to extend it. From a practical point of view, it will usually be easier to prototype mathematical and statistical methods in a "higher-level" language, for example Matlab, R, or Python. Although these languages can be slower to run than optimized code in a

Citation: Osborne JM, Bernabeu MO, Bruna M, Calderhead B, Cooper J, et al. (2014) Ten Simple Rules for Effective Computational Research. PLoS Comput Biol 10(3): e1003506. doi:10.1371/journal.pcbi.1003506

Editor: Philip E. Bourne, University of California San Diego, United States of America

Published March 27, 2014

Funding: This work is part of the 2020 Science programme, which is funded through the EPSRC Cross-Disciplinary Interface Programme (grant number EP/I017909/1) and is also supported by Microsoft Research Ltd. The funders had no role in the preparation of the manuscript.

Competing Interests: ND, SJD, and SE are paid employees of Microsoft Research (MSR) and JMO, MB, RF, and GJM are partly supported by MSR. This does not alter our adherence to the PLOS policies on sharing data and materials. All other authors have declared that no competing interests exist.

* E-mail: James.Osborne@cs.ox.ac.uk

"lower-level" language, their straightforward nature, built-in functionality, and available libraries mean that you will spend less time expressing your ideas in code and searching for bugs.

Rule 3: Make Your Code Understandable to Others (and Yourself)

When revising or adapting existing code, the absence of documentation and comments can result in errors and time drains. Such documentation not only makes your code more understandable to others but also to your future self (put simply, the code tells you "how", the comments tell you "why"). The program code itself can be made more understandable by using meaningful variable names and formatting the code consistently. While commenting and documentation is often neglected when faced with deadlines, developing and maintaining a standardised way of commenting your code will be of great benefit. As well as low-level documentation in the code, you should maintain a record of the "big picture" functionality (i.e., interconnectivity of components and input/output formats). This could take the form of a high-level diagram or description of the system, whether by hand on paper, in verbose code comments, or using standardized approaches such as UML (Unified Modelling Language) (see Text S1). When you are reviewing your code for documentation you should actively seek ways to break it up into modules. This not only aids structure and readability but also avoids the error-prone and tedious task of debugging and updating two (or more) copies of the same code. As a rule of thumb, if you write the same code twice, it should become a function, subroutine, or method.

Rule 4: Don't Underestimate the Complexity of Your Task

When developing your code, you should keep a record of your work. This could be in the form of a "logbook" file or a paper notebook where you store commonly used commands and other notes; another good option is an online tool such as Evernote (http://evernote.com/). You will often find that you have to choose between spending a long time doing a task by hand and possibly spending longer learning how to automate it. In order to automate the task, you will probably need to learn how to use some basic tools such as text editors or scripting languages. Don't be tempted to think, "This is just a one-off, I'll get on with it;" it won't be. You will find bugs, wish to change a parameter, or need to alter a figure slightly, and you will eventually have to repeat the whole process. Even if you are *certain* that it really is a one-off task, use your "logbook" and keep a record of the list of commands you used, since this is the first step towards automating the task if and when the time comes. However, it is not appropriate to automate everything, and you need to find a good balance, automating opportunistically, taking the expected time and cost into account. A good rule to follow is "the rule of three:" once you have had to do the same thing twice already, automate it.

Rule 5: Understand the Mathematical, Numerical, and Computational Methods Underpinning Your Work

When solving any computational model, you should always ensure that you are using the appropriate numerical method for your problem, and that any constraints and conditions are satisfied. A basic understanding of numerical analysis and, in particular, the concepts of rate of convergence, order, and stability of numerical methods will pay dividends. Care should also be taken to ensure that any assumptions made in the derivation of the underlying mathematical models or methods (e.g., having a sufficiently large number of objects to permit a continuum approximation) hold for all system states of interest. You should consult the relevant literature (and communities) that explains these methods and their advantages and/or disadvantages and not steam ahead without first gaining an understanding of which methods are appropriate. By fully understanding the mathematical and numerical methods being used, you can be confident that your results reflect the true behaviour of the underlying model and are not numerical or computational artefacts.

Rule 6: Use Pictures: They Really Are Worth a Thousand Words

Visualisation and graphics are fundamental to developing, understanding, and testing hypotheses, and are indispensable for verifying and validating computational methods (e.g., revealing correlations, covariation, position, structure, flows, orientation, anomalies, and outliers). So, from day one, spend time developing the visual components of your work. Learn, develop and use visualisation software and tools to ensure that you understand your research outputs and can effectively communicate your findings. You may well need to develop novel visualisations for your work, but keep the basic figures. You needed them to understand your results, model, and implementation, and so will anyone else. You should ensure that your visualisation algorithms can be executed separately so that they can be reused by you and others (for the same and different tasks) and refined for other formats (e.g., publications, presentations, and websites). In reality, all scientists could be better educated in design, so any investment will be rewarded, especially by receiving feedback on visualization from users.

Rule 7: Version Control *Everything*

Version control systems (VCSs) offer an easy way to store and back up not only the current version of your code that you are working on but also every previous version of the code (in what's known as a repository). This not only saves you from having to keep multiple copies of the same file but also allows you to "roll back" to an older "working" version of the code if things go wrong. VCSs also allow you to share material between multiple machines, operating systems, and more importantly, users in a simple and robust manner. Two of the most popular VCSs are Subversion (http://subversion.apache.org) and Git (http://www.github.com), both of which offer many advanced features for managing your code. Cloud storage such as Dropbox (http://www.dropbox.com) and SkyDrive (http://www.skydrive.live.com) offer basic file sharing and backup facilities; however, they don't offer the code management features of true VCSs, so the effort put in to learning a VCS is well worth it (see Text S1 for guides on getting started with VCSs). While the primary use of version control is to manage the development and distribution of code, many other collaborative endeavours can be stored in a version control repository. In particular, using version control tools while preparing publications can save time and effort, especially when dealing with input from multiple authors. For example, contributions to this manuscript were managed using a VCS.

Rule 8: Test *Everything*

Any non-trivial computer program will have bugs when first written, often subtle

ones that are hard to detect, which may lead to incorrect results. Indeed, in extreme cases this has caused high-profile retractions of papers [10]. Simple tests that the software behaviour matches expectations are essential for ensuring robust results, minimising the presence of bugs, and gaining confidence in your code (for you and others). As a result of the time pressures inherent in academia, often software testing is performed manually in an *ad hoc* manner, to determine whether results ''look roughly right'' [11]. However, a systematic approach to testing pays dividends. You should learn how to test effectively to avoid the illusion of reliability. For example, compare low-level routines against analytical or prototype solutions (see Rule 2) or experimental data and consider ''corner cases'' and both branches of ''if'' statements. Get the computer to run tests for you automatically and alert you to problems, using a suitable testing framework (see Text S1). Ideally this should be tied to a version control system (see Rule 7) so that tests are run automatically whenever new code is committed to the repository. A useful rule is to turn bugs you fix into new tests to avoid them recurring. Testing gives you the confidence to modify your code without worrying that you are breaking it. Testing can also provide a means for reproducing results of published papers. By setting up a test comparing against published values, you can easily find out when fixing a newly identified bug changes published results.

Rule 9: Share *Everything*

Just as it is a common practice to publish your research findings in peer-reviewed journals, if an important part of your research involves developing new software tools and/or collecting new data, you should consider sharing these [7]. Based on our collective experience, we advocate an open approach of sharing source code, data, and results as freely as possible. You should ask yourself, ''Why not share?'' If the answer is, ''I am worried that people would find mistakes in it,'' then, as a scientist, this should be the strongest argument in favour of sharing it! The provision of such resources openly provides the means to replicate, reproduce, and examine newly developed methods and techniques. Open sharing not only facilitates the scientific enterprise through replication, validation, and error checking, but also deters fraud and malpractice through transparency. It is our opinion that the many arguments in favour of openly sharing code, data, and results far outweigh any against. In many modern computational analyses, the source code represents a readable, executable methodology of the research in question. Sharing is the key to a sustainable future for computational science, and publishers are beginning to require it, with some considering reviewing the software used to generate results [12].

Rule 10: Keep Going!

Our advice arises from our collective experience, and we continue to strive to obey these rules in our work. Scientists have a wide variety of demands on their time (researching, writing papers [13], teaching [14], applying for grants, administration, etc.) and have to make the most of limited resources. Becoming more technically effective can seem daunting without strategies for making progress and keeping motivated. So, prioritise in a way that suits you and your projects and career aspirations. One strategy is to implement another of these rules each time you start a new project, to build a growing repertoire rather than trying to do everything at once. Take every opportunity to teach and help others to do what you have learnt.

Supporting Information

Text S1 Supplementary material for paper. Includes guides for getting started with each rule, along with references to useful links and further reading.
(PDF)

References

1. Kumar S, Dudley J (2007) Bioinformatics software for biologists in the genomics era. Bioinformatics 23: 1713–1717. doi:10.1093/bioinformatics/btm239
2. Karr JR, Sanghvi JC, Macklin DN, Gutschow MV, Jacobs JM, et al. (2012) A Whole-Cell Computational Model Predicts Phenotype from Genotype. Cell 150: 389–401. doi:10.1016/j.cell.2012.05.044
3. Mirams GR, Arthurs CJ, Bernabeu MO, Bordas R, Cooper J, et al. (2013) Chaste: an open source C++ library for computational physiology and biology. PLOS Comput Biol 9: e1002970. doi:10.1371/journal.pcbi.1002970
4. Dalchau N, Phillips A, Goldstein LD, Howarth M, Cardelli L, et al. (2011) A peptide filtering relation quantifies MHC class I peptide optimization. PLOS Comput Biol 7: e1002144. doi:10.1371/journal.pcbi.1002144
5. Bernabeu MO, Nash RW, Groen D, Carver HB, Hetherington J, et al. (2013) Impact of blood rheology on wall shear stress in a model of the middle cerebral artery. Interface Focus 3: 20120094. doi:10.1098/rsfs.2012.0094
6. Mozilla Science Lab (2013) Software Carpentry. Available: http://software-carpentry.org/. Accessed 18 March 2013.
7. Prlić A, Procter JB (2012) Ten simple rules for the open development of scientific software. PLOS Comput Biol 8: e1002802. doi:10.1371/journal.pcbi.1002802
8. Dall'Olio GM, Marino J, Schubert M, Keys KL, Stefan MI, et al. (2011) Ten simple rules for getting help from online scientific communities. PLOS Comput Biol 7: e1002202. doi:10.1371/journal.pcbi.1002202
9. Michaut M (2011) Ten simple rules for getting involved in your scientific community. PLOS Comput Biol 7: e1002232. doi:10.1371/journal.pcbi.1002232
10. Chang G, Roth CB, Reyes CL, Pornillos O, Chen Y, et al. (2006) Retraction. Science 314: 1875. doi:10.1126/science.314.5807.1875b
11. Pitt-Francis J, Bernabeu MO, Cooper J, Garny A, Momtahan L, et al. (2008) Chaste: Using agile programming techniques to develop computational biology software. Phil Trans R Soc A 366: 3111–3136. doi:10.1098/rsta.2008.0096
12. Hayden EC (2013) Mozilla plan seeks to debug scientific code. Nature 501: 472. doi:10.1038/501472a
13. Bourne PE (2005) Ten simple rules for getting published. PLOS Comput Biol 1: e57. doi:10.1371/journal.pcbi.0010057
14. Vicens Q, Bourne PE (2009) Ten simple rules to combine teaching and research. PLOS Comput Biol 5: e1000358. doi:10.1371/journal.pcbi.1000358

Ten Simple Rules for Effective Computational Research: Supplementary Material

Introduction

This supplementary material, for the paper ‘Ten Simple Rules for Effective Computational Research’, is designed to provide more detailed guides to getting started with the rules specified in the paper.

It is broken up into sections, which map onto the rules. Each section contains the following information: a short guide/tips for ‘**Getting started**’; ‘**Useful links**’ to articles and websites of interest for the general reader to help with getting started; and a list of more in-depth references for ‘**Further reading**’.

While we have strived to be as complete as possible we could not hope to provide an exhaustive list of software and links. From our varied interests we have compiled a set of links and references that should serve as a solid starting point.

Rule 1: Look before you leap

Getting started

- Talk to other people in your group to discover the sorts of software they use and whether you can make use of it.
- Be aware of licensing restrictions on software you want to use; the Open Source Initiative (http://opensource.org/licenses/category) and OSS Watch (http://www.oss-watch.ac.uk/) are good websites to consult.
- An article on conducting systematic literature reviews in software engineering:
    - P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain, Journal of Systems and Software, 2007, 80(4), 571–583, http://dx.doi.org/10.1016/j.jss.2006.07.009

Useful links

- Wikipedia provides a list of some established software repositories: http://en.wikipedia.org/wiki/Software_repository
- GitHub (https://github.com/) and SourceForge (http://sourceforge.net/) are two of the most popular software repositories and contain all types of open source software including scientific tools.
- MATLAB (http://www.mathworks.co.uk/products/matlab/) is a widely-used scientific computing environment and programming language developed by MathWorks. While MATLAB itself is commercial software, there is a wide variety of open source code written for MATLAB available online. MATLAB Central is one such repository and is a great place to find MATLAB scripts and functions: http://www.mathworks.co.uk/matlabcentral/.
- R (http://www.r-project.org/) is a free environment for statistical computing and analysis. In addition to its built-in functionality, there are a large number of online repositories of R code. A good place to start is http://cran.r-project.org/.
- If you are writing C++ code then the following libraries may prove useful:
    - Boost (http://www.boost.org/) offers a large number of C++ libraries useful for scientific computing;
    - PETSc (http://www.mcs.anl.gov/petsc/) is a C++ library for performing large-scale matrix manipulation.
- Many libraries exist to extend the functionality of Python. Scientific Python (http://www.scipy.org/) adds numerous linear and nonlinear solvers and matplotlib (http://matplotlib.org/) adds high-quality graphical capabilities.
- Python Package Index (http://pypi.python.org), commonly known as ‘PyPi’, is a good place to find and share Python code.

Further reading

- W.B. Frakes and K. Kang. Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering, 2005, 31(7), 529-536, http://doi.ieeecomputersociety.org/10.1109/TSE.2005.85.

- An interesting blog post on software reuse: http://www.infoq.com/articles/vijay-narayanan-software-reuse.

Rule 2: Develop a prototype first

Getting started

- Consider prototyping your code by implementing a simplified version first, and build up the functionality over several steps.
- While low-level languages such as C++ are undoubtedly faster in the long run, high-level languages such as those listed below offer useful debugging tools and other inbuilt functionality to help speed up prototype development.
- A short blog post guide to prototyping for scientist-programmers: http://www.programming4scientists.com/2008/09/08/prototyping-your-code/

Useful links

- MATLAB (http://www.mathworks.co.uk/): scientific computing environment and programing language based on principles of linear algebra.
- Gnu Octave (http://www.gnu.org/software/octave/): open source equivalent of MATLAB.
- Mathematica (http://www.wolfram.co.uk/mathematica/): environment for performing algebraic manipulation and algebraic solution of various types of equations.
- Maxima (http://maxima.sourceforge.net/): open source algebraic manipulation code.
- R (http://www.r-project.org/): language and environment for performing statistical computing and graphics.

Further reading

- J. Pitt-Francis and J. Whiteley. Guide to Scientific Computing in C++. Springer 2012. Chapter 1. (Note that many of the other rules presented in the paper appear as end of chapter ‘tips’ in this textbook.)

Rule 3: Make your code understandable to others (and yourself)

Getting started

- Making your code easily understood will help you, and others, maintain and debug it. This involves not just the code itself, but also descriptive comments. Consider what you would want to find if you were looking at someone else's code for the first time: your code should tell you *how* something is done and your comments should tell you *why*.
- Start with the basics: name and date each section of code you write (or consider using the date it was last edited). You'll surprise yourself at how useful a date can be.
- Using meaningful variable names greatly improves the readability of code, e.g. var_1 and var_2, versus BirthRate and DeathRate.
- Many languages have coding *conventions* that recommend a particular style for program code, covering comments, indentation, variable naming, etc. These vary between languages (you can found some examples at http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax)#Comments), but learning about these conventions can be useful for reading other people's source code and also for creating your own.

Useful links

- 'Writing Readable Source Code' article from the Software Sustainability Institute: http://software.ac.uk/resources/guides/writing-readable-source-code
- 'Writing Understandable Code' from Dr. Dobbs: http://www.drdobbs.com/writing-understandable-code/184414802
- Wikipedia article on source code comments, covering many languages and other uses of comments: http://en.wikipedia.org/wiki/Comment_(computer_programming)
- Wikipedia article reviewing coding conventions: http://en.wikipedia.org/wiki/Coding_conventions#Common_conventions
- Useful tools for generating documentation include Javadoc and Doxygen, which can produce cross-linked HTML from comments.
    - Doxygen (any language): http://www.doxygen.org/
    - Javadoc (Java): http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
    - Help files in MATLAB: http://www.mathworks.co.uk/help/matlab/matlab_prog/add-help-for-your-program.html
    - Pydoc (Python): http://docs.python.org/2/library/pydoc.html
- Tutorial for describing code using UML: http://www.tutorialspoint.com/uml/uml_building_blocks.htm

Further reading

- R.C. Martin. Clean code: a handbook of agile software craftsmanship. Prentice Hall, 2008.

- ‘Developing Maintainable Software’ from the Software Sustainability Institute: http://software.ac.uk/resources/guides/developing-maintainable-software (discusses the great idea of having your source code reviewed by your peers).
- C. Seiwald. The Seven Pillars of Pretty Code, EETimes 2003: http://eetimes.com/electronics-news/4196975/Seven-Pillars-of-Pretty-Code
- A. Oram and G. Wilson. Beautiful Code, Leading Programmers Explain How They Think. O'Reilly Media 2007.

Rule 4: Don't underestimate the complexity of your task

Getting started

There are some basic computational tools that will pay dividends if they are learnt early, some of which are more specific to coding practices, but some are also useful for the presentation of work:

- The Linux command line, as well as a command line text editor such as *emacs* or *vi*.
- Simple text file processing tools, such as *sed* and *awk*, and *grep* for searching for phrases in files (useful when you are debugging).
- Scripting languages such as *bash* (or more usefully *perl* or *python*).
- A build utility such as *make*, *CMake* or *SCons*.
- A job scheduler (if you don't want to be sat at your computer overnight!) such as *cron*.
- LaTeX for presenting mathematics in written documents.
- A literature reference (bibliography) manager.
- And of course the appropriate manual and help pages for the packages you are using.

Useful links

- List of useful Linux command lines: http://www.pixelbeat.org/cmdline.html.
- A useful introduction to LaTeX for beginners: http://www.andy-roberts.net/writing/latex.

Further reading

- The act of reviewing your code, looking for repeated functionality, and moving it into useful functions is commonly known as The Rule of Three. It was introduced in
    - M. Fowler, K. Beck, J. Brant, and W Opdyke. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.
- There is a wealth of books about simple tools. An example is
    - C. Albing and J.P. Vossen. Bash Cookbook: Solutions and Examples for Bash Users. O'Reilly, 2007.

Rule 5: Understand the mathematical, numerical and computational methods underpinning your work

Getting started

The development of any computational method should involve a number of important issues:

- Ensure that your scientific approach is sound (modelling assumptions, suitability of analyses)
- Ensure that your chosen methods are accurate and stable
- Ensure that your implementation is efficient and bug free

While balancing these can be time-consuming and tricky, these issues underpin the 'correctness' of your approach and you should be aware of them in the development of your models and programs. For example, testing (Rule 8) can be useful for checking that you've implemented an algorithm correctly (or the library you're using has) by validating results with some known solutions.

Useful links

- MIT Open Courseware in Numerical Analysis: http://ocw.mit.edu/courses/mathematics/18-03sc-differential-equations-fall-2011/unit-i-first-order-differential-equations/numerical-methods/
- Consider profiling to quantify the computational cost of different parts of your programs: http://en.wikipedia.org/wiki/Profiling_(computer_programming)
- The Software Carpentry lesson on algorithmic complexity: http://www.software-carpentry.org/4_0/invperc/tuning.html

Further reading

- K.W. Morton and D.F. Mayers. Numerical Solution of Partial Differential Equations. Cambridge University Press, 2005.
- G.H.Golub and C.F. Van Loan. Matrix Computations (3rd ed.). John Hopkins University Press, 1996.
- W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Numerical Recipes in C++: The Art of Scientific Computing (3rd ed.). Cambridge University Press, 2007.
- T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms (3rd ed.). MIT Press, 2009.
- D.E. Knuth. The Art of Computer Programming (multiple volumes). Addison Wesley, 2011.
- R. Gerber.The Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture. Intel Press, 2002.
- A. Fog. Optimizing software in C++: http://www.agner.org/optimize/optimizing_cpp.pdf
- P. Getreuer. Writing Fast MATLAB code: http://www.mathworks.com/matlabcentral/fileexchange/5685

6: Use pictures: they really are worth a thousand words

Getting started

- Investing more time and effort in visualisation and graphics can enhance everything you do - from exploring data and debugging code/models all the way to conference presentations - so reassess the amount of effort and time you will invest. Find some galleries of useful visualisations/graphics and reconsider what you could achieve.
- From the outset think about what you want to achieve and what the visualizations purpose is. Who is it for? What is the key message? Writing a design brief (even quickly) can formalise these goals and provide a set of requirements that you can evaluate your visualisation by. Consider the audience and end-users at the start.
- Understand your workflow and what role your visualisations and graphics will have. Is it a throw-away graph to check for outliers in the data? Or will it really serve a purpose later on as well, e.g. in a presentation, software, a publication, or tutorial?
- Start with pen and paper before you start coding a visualisation and try different variations for the main axes of the figure, and the secondary axes. Understanding and thinking about the visual encodings you will use will save time later. If you can't say way you have made one design choice over another then go back to the drawing board.
- When implementing your design ensure that you write the visualisation as a function so that it can be re-used with any data set, and so any changes are easy to implement. For instance, if you are editing a paper and want to change the colour of some points you don't want to have to run all your models again. You should be able to supply all the data and code for a publication's figures as independent units.
- Treat you visualisations in the same way as you would text and test it out on your colleagues, friends and family. If it requires a PhD to read and understand your visualisations then you may need to go back to the drawing board. 'Cool' graphics serve a different purpose to 'informative' graphics so make sure that you are true to your design brief.

Useful links

Give your training in visualisation a reboot and read:

- A recent perspective on visualisation for biological data: S.I. O'Donoghue, A.C. Gavin, N. Gehlenborg, D.S. Goodsell, J.K. Heriche, C.B. Nielsen, C. North, A.J. Olson, J.B. Procter, D.W. Shattuck, T. Walter, and B. Wong. Visualizing biological data-now and in the future. Nature Methods. 2007. 7:S2-4. http://dx.doi.org/10.1038/nmeth.f.301
- A monthly column on design for scientific graphics: Wong, B. Points of view: Color coding. Nature Methods. 2010. 7, 573. http://dx.doi.org/10.1038/nmeth0810-573

Explore what tools are available and consider using a new tool to fit your goals:

- http://processing.org/ - '**Processing** is an open source programming language and environment for people who want to create images, animations, and interactions.'
- http://www.paraview.org/ - '**ParaView** is an open-source, multi-platform data analysis and visualization application. ParaView users can quickly build visualizations to analyze their data using qualitative and quantitative techniques. The data exploration can be done interactively in 3D or programmatically using ParaView's batch processing capabilities.'

- http://d3js.org/ - '**D3.js** is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.'
- http://research.microsoft.com/en-us/um/cambridge/groups/science/tools/datasetviewer/datasetviewer.htm - '**DataSet Viewer** is a simple standalone menu-driven tool for quickly exploring and comparing time series, geographic distributions and other patterns within scientific data. DataSet Viewer combines selection, filtering and slicing tools, with various chart types (scatter plots, line graphs, heat maps, as well as tables), and geographic mapping (using Bing Maps). The resulting views can be exported as images or movies, or bundled into an interactive package that be shared with colleagues. '

Further reading

- A book about visualisation in practice and the 'Processing' visualisation language: B. Fry. Visualizing data. O'Reilly Media Inc, 2008. http://shop.oreilly.com/product/9780596514556.do
- A 'design' book about networks visualisations full of ideas: M. Lima. Visual Complexity: Mapping Patterns of Information. Princeton Academic Press, 2011. http://www.visualcomplexity.com/vc/
- An information visualisation blog - http://eagereyes.org/about
- Find some of Murrell's book on Graphics and plotting in R and code for his examples at http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html
- Listen to the http://datastori.es/ podcast to hear about issues and solutions in data visualisation.
- If you can muster the cash, go to a visualisation conference: http://visweek.org/

Rule 7: Version control everything

Getting started

- Understand the capabilities of version control and how it can help you in your everyday work and how it can make collaborative work less painful. For that, we recommend watching this introductory video from the software carpentry course. http://software-carpentry.org/4_0/vc/intro.html
- Learn the jargon! Version control systems (VCS) are based on very simple ideas, but might seem overwhelming to the novice as they use very specific jargon such as commit, checkout, or cherry-pick. A good source to learn the words can be found in, 'A Visual guide to Version Control' (http://betterexplained.com/articles/a-visual-guide-to-version-control/). Also, be careful that the same terms can mean different things in different VCS! For example, 'merge' doesn't mean the same in git as it does in mercurial.
- Choose your version control system (even though there are several systems available which you may want to learn to use and alternate in the future, we advise you focus on one to start with). Before you can make this choice, it is crucial to understand the differences between a centralised and a distributed VCS (well explained here http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/). A good starting point for online repositories and open-source projects is http://github.com (and the techniques used here can later be transferred to other systems).
- Note that when writing documents in LaTeX using version control you could try start each sentence on a new line as this will make tracking changes much easier.

Useful links

Line-by-line version control tools - store the entire history of changes, and allows conflict resolution. Should be used for any type of code development or shared work.

- Subversion (SVN): http://subversion.apache.org/
    - Assembla: https://www.assembla.com/home a free SVN repository host.
- Git: www.github.com Good links to learn how to use Git are:
    - Git Immersion: 'A guided tour that walks through the fundamentals of git, inspired by the premise that to know a thing is to do it.': www.http://gitimmersion.com/
    - A Visual Git Reference: 'This page gives brief, visual reference for the most common commands in git.': http://marklodato.github.com/visual-git-guide/index-en.html
    - This memrise course is a very fun way to practice and memorise the key commands of Git: http://www.memrise.com/course/63157/git-commands-2/
- Mercurial: http://mercurial.selenic.com/

Also, even though not VCS as such (or with full VCS capabilities), the examples below are great for managing folders of documents, presentations, or visualisations that only you work on, or when you take turns with your collaborators.

- SkyDrive: http://www.skydrive.live.com
- Dropbox: http://www.dropbox.com
- Google Drive: https://drive.google.com

- Office 365: http://office365.microsoft.com/

Further reading

Version Control by Example: http://www.ericsink.com/vcbe/

Rule 8: Test everything

Getting started

- Test everything means test everything. Your code should be written in short logical parts (Rules 3 and 4). On this basis each single function should be tested separately and in combination with the others.
- Think about the oddest inputs and test all of them. Test all extreme cases. What would someone do just to make your program fail? Try to make your program fail!
- Think about writing your test cases before you actually start writing your program. At this time you will be way more motivated to write extensive test cases than afterwards. This approach will also help you to plan your program wisely.
- Find a tool for testing that you like to use (see links below) or design the testing environment by yourself. Either way test your code.

Useful links

- Overview about testing: http://software-carpentry.org/4_0/test/.
- Glossary for terms used in testing: http://www.origsoft.com/whitepapers/software-testing-glossary/glossary_of_terms.pdf.
- List of testing frameworks: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Further reading

- Z. Merali. Why Scientific Programming Does Not Compute. Nature, 2010. 467:775–777. http://dx.doi.org/10.1038/467775a
- R. Sanders, D. Kelly. The Challenge of Testing Scientific Software. Proceedings Conference for the Association for Software Testing (CAST), Toronto, 2008, pages 30-36: http://www.cacr.caltech.edu/projects/danse/doc/kelly-sanders-01.pdf
- M. Feathers. Working Effectively with Legacy Code. Prentice Hall, 2004. http://rads.stackoverflow.com/amzn/click/0131177052
- B. Meyer. Seven Principles of Software Testing. Computer, 2008. vol.41, no.8, pages 99-101, http://dx.doi.org/10.1109/MC.2008.306

Rule 9: Share everything

Getting started

- Openness and sharing are now taken to be an important aspect of the scientific enterprise. For example, see the report on the uptake of open science: Science as an open enterprise The Royal Society Science Policy Centre report 02/2012: http://royalsociety.org/events/2012/science-open-enterprise/
- If open dissemination of your software/data is likely, addressing this from the beginning will save time later. For example, place your outputs in a public version controlled repository like GitHub (c.f. rule 7).
- A recent paper focusing on developing open source software: A. Prlić, J.B. Procter. Ten Simple Rules for the Open Development of Scientific Software. PLoS Comput Biol 2012 8(12): e1002802. http://dx.doi.org/10.1371/journal.pcbi.1002802

Useful links

- A blog on developing and releasing free and open source software: http://www.openscience.org/.
- An open repository for the deposition and sharing of protocols for scientific research: http://www.nature.com/protocolexchange/.
- Dryad: an international repository of data underlying peer-reviewed articles in the basic and applied biosciences: http://datadryad.org/.
- The Open Source Initiative: advocate of open source software: http://opensource.org/.
- The Open Data Institute: http://www.theodi.org/.
- Figshare: a repository where users can make all of their research outputs, in any file format, available in a citable, sharable and discoverable manner: http://figshare.com.

Further reading

- Links to advice on copyright issues:
    - http://toolkit.vph-noe.eu/component/docman/doc_download/1-g07-toolkit-licensing-guideline-v10
    - http://www.oss-watch.ac.uk/resources/iprguide
- D. Ince, L. Hatton, and J. Graham-Cumming. The case for open computer programs. Nature 2012. Volume 482 pages 485-488. http://dx.doi.org/doi:10.1038/nature10836