Piccolo and Frampton *GigaScience* (2016) 5:30
DOI 10.1186/s13742-016-0135-4

GigaScience

REVIEW Open Access

Tools and techniques for computational reproducibility

Stephen R. Piccolo[1*] and Michael B. Frampton[2]

Abstract

When reporting research findings, scientists document the steps they followed so that others can verify and build upon the research. When those steps have been described in sufficient detail that others can retrace the steps and obtain similar results, the research is said to be reproducible. Computers play a vital role in many research disciplines and present both opportunities and challenges for reproducibility. Computers can be programmed to execute analysis tasks, and those programs can be repeated and shared with others. The deterministic nature of most computer programs means that the same analysis tasks, applied to the same data, will often produce the same outputs. However, in practice, computational findings often cannot be reproduced because of complexities in how software is packaged, installed, and executed—and because of limitations associated with how scientists document analysis steps. Many tools and techniques are available to help overcome these challenges; here we describe seven such strategies. With a broad scientific audience in mind, we describe the strengths and limitations of each approach, as well as the circumstances under which each might be applied. No single strategy is sufficient for every scenario; thus we emphasize that it is often useful to combine approaches.

Keywords: Computational reproducibility, Practice of science, Literate programming, Virtualization, Software containers, Software frameworks

Background

When reporting research, scientists document the steps they followed to obtain their results. If the description is comprehensive enough that they and others can repeat the procedures and obtain semantically consistent results, the findings are considered to be "reproducible" [1–6]. Reproducible research forms the basic building blocks of science, insofar as it allows researchers to verify and build on each other's work with confidence.

Computers play an increasingly important role in many scientific disciplines [7–10]. For example, in the United Kingdom, 92 % of academic scientists use some type of software in their research, and 69 % of scientists say their research is feasible only with software tools [11]. Thus efforts to increase scientific reproducibility should consider the ubiquity of computers in research.

Computers present both opportunities and challenges for scientific reproducibility. On one hand, the deterministic nature of most computer programs means that identical results can be obtained from many computational analyses applied to the same input data [12]. Accordingly, computational research can be held to a high reproducibility standard. On the other hand, even when no technical barrier prevents reproducibility, scientists often cannot reproduce computational findings because of complexities in how software is packaged, installed, and executed—and because of limitations associated with how scientists document these steps [13]. This problem is acute in many disciplines, including genomics, signal processing, and ecological modeling [14–16], which have large data sets and rapidly evolving computational tools. However, the same problem can affect any scientific discipline requiring computers for research. Seemingly minor differences in computational approaches can have major influences on analytical outputs [12, 17–22], and the effects of these differences may exceed those resulting from experimental factors [23].

Journal editors, funding agencies, governmental institutions, and individual scientists have increasingly made calls for the scientific community to embrace practices to support computational reproducibility [24–31]. This

* Correspondence: stephen_piccolo@byu.edu
Twitter: @stevepiccolo
[1]Department of Biology, Brigham Young University, Provo, UT 84602, USA
Full list of author information is available at the end of the article

BioMed Central

© 2016 The Author(s). **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated.

movement has been motivated, in part, by scientists' failed efforts to reproduce previously published analyses. For example, Ioannidis et al. evaluated 18 published research studies that used computational methods to evaluate gene expression data, but they were able to reproduce only two of those studies [32]. In many cases, the culprit was a failure to share the study's data; however, incomplete descriptions of software-based analyses were also common. Nekrutenko and Taylor examined 50 papers that analyzed next-generation sequencing data and observed that fewer than half provided any details about software versions or parameters [33]. Recreating analyses that lack such details can require hundreds of hours of effort [34] and may be impossible, even after consulting the original authors. Failure to reproduce research may also lead to careerist effects, including retractions [35].

Noting such concerns, some journals have emphasized the value of placing computer code in open access repositories. It is most useful when scientists provide direct access to an archived version of the code via a uniform resource locator (URL). For example, Zenodo.org and figshare.com provide permanent digital object identifiers (DOI) that can link to software code (and other digital objects) used in publications. In addition, some journals have extended requirements for "Methods" sections, now asking researchers to provide detailed descriptions of 1) how to install software and its dependencies, and 2) what parameters and data preprocessing steps are used in analyses [10, 24]. A 2012 Institute of Medicine report emphasized that, in addition to computer code and research data, "fully specified computational procedures" should be made available to the scientific community [25]. The report's authors elaborated that such procedures should include "all of the steps of computational analysis", and that "all aspects of the analysis need to be transparently reported" [25]. Such policies represent important progress. However, it is ultimately the responsibility of individual scientists to ensure that others can verify and build upon their analyses.

Describing a computational analysis sufficiently—such that others can re-execute, validate, and refine it—requires more than simply stating what software was used, what commands were executed, and where to find the source code [13, 27, 36–38]. Software is executed within the context of an operating system (for example, Windows, Mac OS, or Linux), which enables the software to interface with computer hardware. In addition, most software relies on a hierarchy of software dependencies, which perform complementary functions and must be installed alongside the main software tool. One version of a given software tool or dependency may behave differently or have a different interface than another version of the same software. In addition, most analytical software offers a range of parameters (or settings) that the user can specify. If any of these variables differs from those used by the original experimenter, the software may not execute properly or analytical outputs may differ considerably from those observed by the original experimenter.

Scientists can use various tools and techniques to overcome these challenges and to increase the likelihood that their computational analyses will be reproducible. These techniques range in complexity from simple (e.g., providing written documentation) to advanced (e.g., providing a virtual environment that includes an operating system and all the software necessary to execute the analysis). This review describes seven strategies across this spectrum. We describe many of the strengths and limitations of each approach, as well as the circumstances under which each might be applied. No single strategy will be sufficient for every scenario; therefore, in many cases, it will be most practical to combine multiple approaches. This review focuses primarily on the computational aspects of reproducibility. The related topics of empirical reproducibility, statistical reproducibility, data sharing, and education about reproducibility have been described elsewhere [39–46]. We believe that with greater awareness and understanding of computational reproducibility techniques, scientists—including those with limited computational experience—will be more apt to perform computational research in a reproducible manner.

Narrative descriptions are a simple but valuable way to support computational reproducibility

The most fundamental strategy for enabling others to reproduce a computational analysis is to provide a detailed, written description of the process. For example, when reporting computational results in a research article, authors customarily provide a narrative that describes the software they used and the analytical steps they followed. Such narratives can be invaluable in enabling others to evaluate the scientific approach and to reproduce the findings. In many situations—for example, when software execution requires user interaction or when proprietary software is used—narratives are the only feasible option for documenting such steps. However, even when a computational analysis uses open-source software and can be fully automated, narratives help others understand how to re-execute an analysis.

Although most articles about research that uses computational methods provide some type of narrative, these descriptions often lack sufficient detail to enable others to retrace those steps [32, 33]. Narrative descriptions should indicate the operating system(s), software dependencies, and analytical software that were used, and how to obtain them. In addition, narratives should indicate the exact software versions used, the order in

which they were executed, and all non-default parameters that were specified. Such descriptions should account for the fact that computer configurations can differ vastly, even for computers with the same operating system. Because it can be difficult for scientists to remember such details after the fact, it is best to record this information throughout the research process, rather than at the time of manuscript preparation [8].

The following sections describe techniques for automating computational analyses. These techniques can diminish the need for scientists to write narratives. However, because it is often impractical to automate all computational steps, we expect that, for the foreseeable future, narratives will play a vital role in enabling computational reproducibility.

Custom scripts and code can automate research analysis

Scientific software can often be executed in an automated manner via text-based commands. Using such commands—via a command-line interface—scientists can indicate the software program(s) to be executed and which parameter(s) should be used. When multiple commands must be executed, they can be compiled into scripts specifying the order in which the commands should be executed (Fig. 1; Additional file 1). In many cases, scripts also include commands for installing and configuring software. Such scripts serve as valuable documentation not only for individuals who wish to re-execute the analysis, but also for the researcher who performed the original analysis [47]. In these cases, no amount of narrative is an adequate substitute for providing the actual commands that were used.

When writing command-line scripts, it is essential to explicitly document any software dependencies and input data that are required for each step in the analysis. The Make utility [48, 49] provides one way to specify such requirements [36]. Before any command is executed, Make verifies that each documented dependency is available. Accordingly, researchers can use Make files (scripts) to specify a full hierarchy of operating system components and dependent software that must be present to perform the analysis (Fig. 2; Additional file 2). In addition, Make can automatically identify any commands that can be executed in parallel, potentially reducing the amount of time required for the analysis. Although Make was originally designed for UNIX-based operating systems (such as Mac OS or Linux), similar utilities have since been developed for Windows operating systems [50]. Table 1 lists various utilities that can be used to automate software execution.

As well as creating scripts to execute existing software, many researchers also create new software by writing computer code in a programming language such as Python, C++, Java, or R. Such code may perform relatively simple tasks, such as reformatting data files or invoking third-party software. In other cases, computer code may constitute a manuscript's key intellectual contribution.

Whether analysis steps are encoded in scripts or as computer code, scientists can support reproducibility by publishing these artifacts alongside research papers. By doing so, authors enable readers to evaluate the analytical approach in full detail and to extend the analysis more readily [51]. Although scripts and code may be included alongside a manuscript as supplementary material, a better alternative is to store them in a public repository with a permanent URL. It is often also useful to store code in a version control system (VCS) [8, 9, 47], and to share it via Web-based services like GitHub.com or Bitbucket.org [52]. With such a VCS

```
#!/bin/bash

# Download software, reference genome, and FASTQ files
wget http://downloads.sourceforge.net/project/bio-bwa/bwakit/bwakit-0.7.12_x64-
linux.tar.bz2
wget -O refGenomeFiles.zip https://ndownloader.figshare.com/files/4841809
wget -O FASTQ.zip https://ndownloader.figshare.com/articles/3114454/versions/1

# Extract software and reference genome files
tar -jxvf bwakit-0.7.12_x64-linux.tar.bz2
unzip refGenomeFiles.zip
unzip FASTQ.zip

# Align FASTQ data to reference genome and save to SAM file
bwa.kit/bwa mem  KJ660346.fa SRR1972917_1.fastq SRR1972917_1.fastq >
SRR1972917_aligned.sam

# Convert SAM file to BAM file
bwa.kit/samtools view -bS SRR1972917_aligned.sam > SRR1972917_aligned.bam

# Sort BAM file
bwa.kit/samtools sort SRR1972917_aligned.bam SRR1972917_aligned_sorted

# Index BAM file
bwa.kit/samtools index SRR1972917_aligned_sorted.bam
```

Fig. 1 Example of a command line script. This script can be used to align DNA sequence data to a reference genome. First, it downloads the software and data files necessary for the analysis. Then, it extracts ("unzips") these files, and aligns the data to a reference genome for Ebola virus. Finally, it converts, sorts, and indexes the aligned data. See Additional file 1 for an executable version of this script

```
all: SRR1972917_aligned_sorted.bam

SRR1972917_aligned_sorted.bam: SRR1972917_aligned.sam
        # Convert SAM file to BAM file
        bwa.kit/samtools view -bS SRR1972917_aligned.sam > SRR1972917_aligned.bam

        # Sort BAM file
        bwa.kit/samtools sort SRR1972917_aligned.bam SRR1972917_aligned_sorted

        # Index BAM file
        bwa.kit/samtools index SRR1972917_aligned_sorted.bam

SRR1972917_aligned.sam: bwa.kit/bwa KJ660346.fa SRR1972917_1.fastq
        # Align FASTQ data to reference genome and save to SAM file
        bwa.kit/bwa mem  KJ660346.fa SRR1972917_1.fastq SRR1972917_1.fastq > SRR1972917_aligned.sam

bwa.kit/bwa:
        # Download and unpack BWA software
        wget http://downloads.sourceforge.net/project/bio-bwa/bwakit/bwakit-0.7.12_x64-linux.tar.bz2
        tar -jxvf bwakit-0.7.12_x64-linux.tar.bz2

KJ660346.fa:
        # Download and unzip reference genome
        wget -O refGenomeFiles.zip https://ndownloader.figshare.com/files/4841809
        unzip refGenomeFiles.zip

SRR1972917_1.fastq:
        # Download and unzip FASTQ files
        wget -O FASTQ.zip https://ndownloader.figshare.com/articles/3114454/versions/1
        unzip FASTQ.zip

clean:
        rm -rfv KJ660346.fa* FASTQ.zip refGenomeFiles.zip bwa.kit* bwakit-0.7.12_x64-linux.tar.bz2 SRR1972917*
```

Fig. 2 Example of a Make file. This file performs the same function as the command line script shown in Fig. 1, except that it is formatted for the Make utility. Accordingly, it is structured so that specific tasks must be executed before other tasks, in a hierarchical manner. See Additional file 2 for an executable version of this file

repository, scientists can track the different versions of scripts and code that have been developed throughout the evolution of the research project. In addition, outside observers can see the full version history, contribute revisions to the code, and reuse the code for their own purposes [53]. When submitting a manuscript, the authors may "tag" a specific version of the repository that was used for the final analysis described in the manuscript.

Table 1 Utilities that can be used to automate software execution

- GNU Make and Make for Windows: tools for building software from source files and for ensuring that the software's dependencies are met.
- Snakemake [109]: an extension of Make that provides a more flexible syntax and makes it easier to execute tasks in parallel.
- BPipe [110]: a tool that provides a flexible syntax for users to specify commands to be executed; it maintains an audit trail of all commands that have been executed.
- GNU Parallel [111]: a tool for executing commands in parallel across one or more computers.
- Makeflow [112]: a tool that can execute commands simultaneously on various types of computer architectures, including computer clusters and cloud environments.
- SCONS [113]: an alternative to GNU Make that enables users to customize the process of building and executing software using scripts written in the Python programming language.
- CMAKE.org: a tool that enables users to execute Make scripts more easily on multiple operating systems.

Software frameworks enable easier handling of software dependencies

Virtually all computer scripts and code relies on external software dependencies and operating system components. For example, suppose a research study required a scientist to apply Student's *t*-test. Rather than write code to implement this statistical test, the scientist would likely find an existing software library that implements the test and then invoke that library from their code. Much time can be saved with this approach, and a wide range of software libraries are freely available. However, software libraries change frequently; invoking the wrong version of a library may result in an error or an unexpected output. Thus, to enable others to reproduce an analysis, it is critical to indicate which dependencies (and versions thereof) must be installed.

One way to address this challenge is to build on a preexisting software framework, which makes it easier to access software libraries that are commonly used to perform specific types of analysis task. Typically, such frameworks also make it easier to download and install software dependencies, and to ensure that the versions of software libraries and their dependencies are compatible with each other. For example, Bioconductor [54], created for the R statistical programming language [55], is a popular framework that contains hundreds of software packages for analyzing biological data. The

Bioconductor framework facilitates versioning, documenting, and distributing code. Once a software library has been incorporated into Bioconductor, other researchers can find, download, install, and configure it on most operating systems with relative ease. In addition, Bioconductor installs software dependencies automatically. These features ease the process of performing an analysis, and can help with reproducibility. Various software frameworks exist for other scientific disciplines [56–61]. General purpose tools for managing software dependencies also exist, for example, Apache Ivy [62] and Puppet [50].

To best support reproducibility, software frameworks should make it easy for scientists to download and install previous versions of a software tool, as well as previous versions of dependencies. Such a design enables other scientists to reproduce analyses that were conducted with previous versions of a software framework. In the case of Bioconductor, considerable extra work may be required to install specific versions of Bioconductor software and their dependencies. To overcome these limitations, scientists may use a software container or virtual machine to package together the specific versions they used in an analysis. Alternatively, they might use third-party solutions such as the aRchive project [63].

Literate programming combines narratives with code

Although narratives, scripts, and computer code individually support reproducibility, there is additional value in combining these entities. Even though a researcher may provide computer code alongside a research paper, other scientists may have difficulty interpreting how the code accomplishes specific tasks. A longstanding way to address this problem is via code comments: human-readable annotations interspersed throughout computer code. However, code comments and other types of documentation often become outdated as code evolves throughout the analysis process [64]. One way to overcome this problem is to use a technique called literate programming [65]. In this approach, the scientist writes a narrative of the scientific analysis and intermingles code directly within the narrative. As the code is executed, a document is generated that includes the code, narratives, and any outputs (e.g., figures, tables) of the code. Accordingly, literate programming helps ensure that readers understand exactly how a particular research result was obtained. In addition, this approach motivates the scientist to keep the target audience in mind when performing a computational analysis, rather than simply to write code that a computer can parse [65]. Consequently, by reducing barriers of understanding among scientists, literate programming can help to engender greater trust in computational findings.

One popular literate programming tool is Jupyter [66]. Using Jupyter.org's Web-based interface, scientists can create interactive "notebooks" that combine code, data, mathematical equations, plots, and rich media [67]. Originally known as IPython, and previously designed exclusively for the Python programming language, Jupyter now makes it possible to execute code in many different programming languages. Such functionality may be important to scientists who prefer to combine the strengths of different programming languages.

knitr [68] has also gained considerable popularity as a literate programming tool. It is written in the R programming language, and thus can be integrated seamlessly with the array of statistical and plotting tools available in that environment. However, like Jupyter, knitr can execute code written in multiple programming languages. Commonly, knitr is applied to documents that have been authored using RStudio [69], an open-source tool with advanced editing and package management features.

Jupyter notebooks and knitr reports can be saved in various output formats, including hypertext markup language (HTML) and portable document format (PDF; see examples in Figs. 3 and 4; Additional files 3 and 4). Increasingly, scientists include such documents as supplementary materials to journal manuscripts, enabling others to repeat analysis steps and recreate manuscript figures [70–73].

Scientists typically use literate programming tools for data analysis tasks that can be executed interactively, in a modest amount of time (e.g., minutes or hours). However, it is possible to execute Jupyter or knitr at the command line; thus longer running tasks can be executed on high-performance computers.

Literate programming notebooks are suitable for research analyses that require a modest amount of computer code. For analyses needing larger amounts of code, more advanced programming environments may be more suitable, perhaps in combination with a "literate documentation" tool such as Dexy.it.

Workflow management systems enable software to be executed via a graphical user interface

Writing computer scripts and code seems daunting to many researchers. Although various courses and tutorials are helping to make this task less formidable [46, 74–76], many scientists use "workflow management systems" to facilitate the execution of scientific software [77]. Typically managed via a graphical user interface, workflow management systems enable scientists to upload data and process them using existing tools. For multistep analyses, the output from one tool can be used as input to additional tools, resulting in a series of commands known as a workflow.

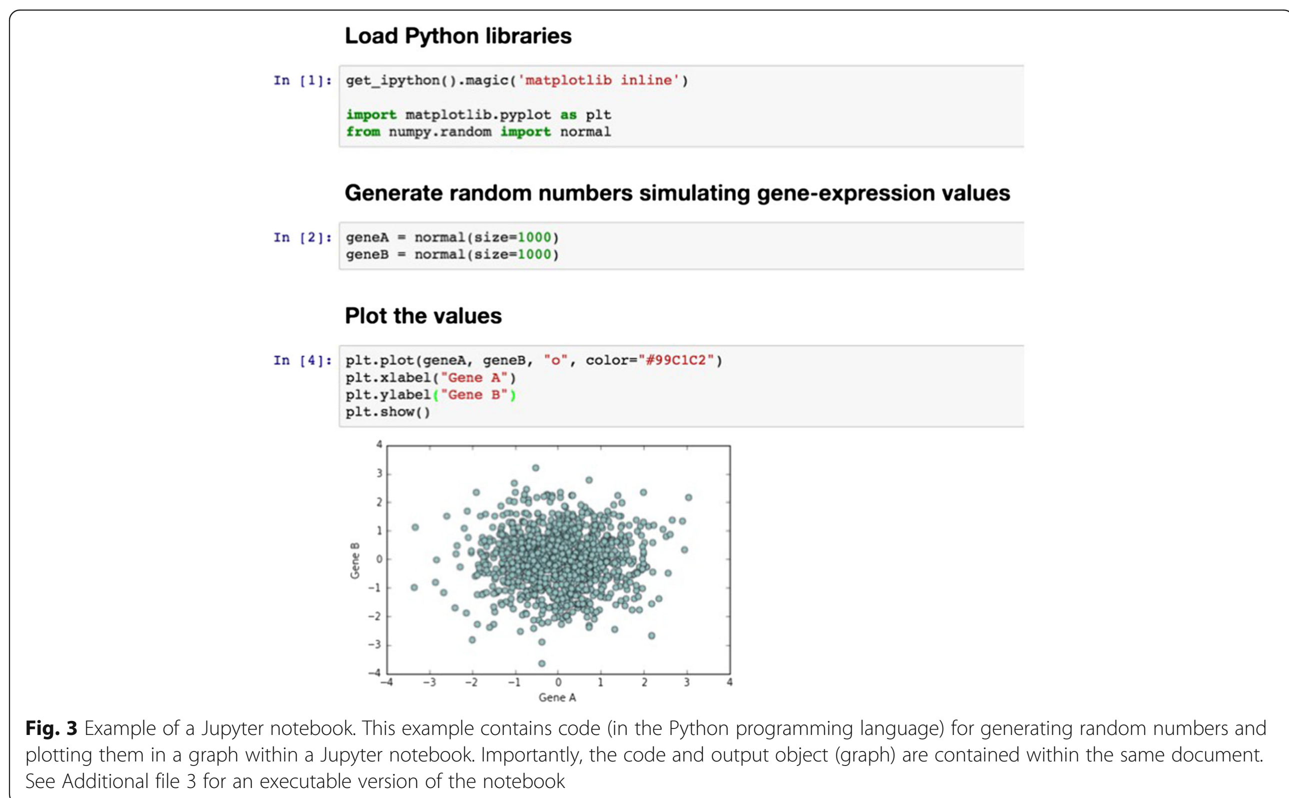

Fig. 3 Example of a Jupyter notebook. This example contains code (in the Python programming language) for generating random numbers and plotting them in a graph within a Jupyter notebook. Importantly, the code and output object (graph) are contained within the same document. See Additional file 3 for an executable version of the notebook

Galaxy [78, 79] has gained considerable popularity within the bioinformatics community, especially for performing next-generation sequencing analysis. As users construct workflows, Galaxy provides descriptions of how software parameters should be used, examples of how input files should be formatted, and links to relevant discussion forums. To help with processing large data sets and computationally complex algorithms, Galaxy also provides an option to execute workflows on cloud-computing services [80]. In addition, researchers can share workflows with each other at UseGalaxy.org; this feature has enabled the Galaxy team to build a community that encourages reproducibility, helps define best practices, and reduce the time required for novices to get started.

Various other workflow systems are freely available to the research community (see Table 2). For example, VisTrails.org is used by researchers from many disciplines, including climate science, microbial ecology, and quantum mechanics [81]. It enables scientists to design visual workflows, and connect data inputs with analytical modules and the resulting outputs. In addition, VisTrails tracks a full history of how each workflow was created. This capability, referred to as "retrospective provenance", makes it possible for others to not only reproduce the final version of an analysis, but also to examine previous incarnations of the workflow and how each change influenced the analytical outputs [82].

Although workflow management systems offer many advantages, users must accept tradeoffs. For example, although the teams that develop these tools often provide public servers where users can execute workflows, many scientists share these resources, limiting the computational power or storage space available to execute large-scale analyses in a timely manner. As an alternative, many scientists install these systems on their own computers; however, configuring and supporting them requires time and expertise. In addition, if a workflow tool does not yet provide a module to support a given analysis, the scientist must create one. This task constitutes additional overheads; however, utilities such as the Galaxy Tool Shed [83] are helping to facilitate this process.

Virtual machines encapsulate an entire operating system and software dependencies

Whether within a literate programming notebook, or via a workflow management system, an operating system and relevant software dependencies must be installed before an analysis is executed. The process of identifying, installing, and configuring such dependencies consumes a considerable amount of scientists' time. Different operating systems (and versions thereof) may require different installation and configuration steps. Furthermore, earlier versions of software dependencies, which may currently be installed on a given computer,

R Markdown Example

Generate random numbers simulating gene-expression values

```
geneA <- rnorm(1000)
geneB <- rnorm(1000)
```

Plot the numbers as a histogram

```
# Set the margins so there won't be too much white space
par(mar=c(4.1, 4.1, 0.1, 0.1))

plot(geneA, geneB, col="#99C1C2", xlab="Gene A", ylab="Gene B", main="")
```

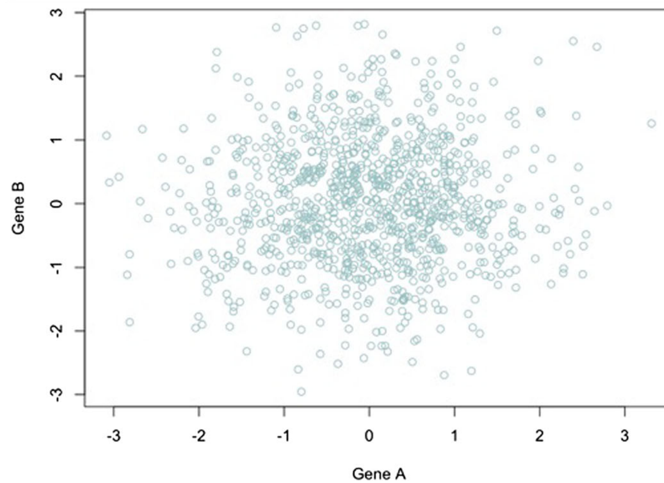

Fig. 4 Example of a document created using knitr. This example contains code (in the R language) for generating random numbers and plotting them on a graph. The knitr tool was used to generate the document, which combines the code and the output object (figure). See Additional file 4 for an executable version of this document

may be incompatible with—or produce different outputs than—newer versions.

One solution is to use virtual machines, which can encapsulate an entire operating system and all software, scripts, code, and data necessary to execute a computational analysis [84, 85] (Fig. 5). Using virtualization software such as VirtualBox or VMWare (see Table 3), a virtual machine can be executed on practically any desktop, laptop, or server, irrespective of the main ("host") operating system on the computer. For example, even though a scientist's computer may be running a Windows operating system, they may perform an analysis on a Linux operating system that is running concurrently—within a virtual machine—on the

Table 2 Workflow management tools freely available to the research community

- Galaxy [78, 79]
- VisTrails [81]
- Kepler-project.org [114]
- CyVerse.org (formerly known as The iPlant Collaborative) [115]
- GenePattern [116–118]
- Taverna.org.uk [119]
- LONI Pipeline [120, 121]

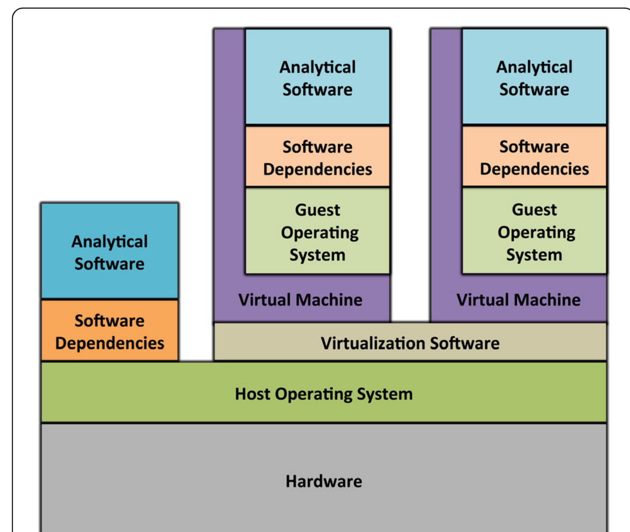

Fig. 5 Architecture of virtual machines. Virtual machines encapsulate analytical software and dependencies within a "guest" operating system, which may be different to the main ("host") operating system. A virtual machine executes in the context of virtualization software, which runs alongside other software installed on the computer

Table 3 Virtual machine software

Virtualization hypervisors:
• VirtualBox.org (open source)
• XenProject.org (open source)
• VMWare.com (partially open source)
Virtual machine management tools:
• VagrantUP.com (open source)
• Vortex (open source) [122]

same computer. The scientist has full control over the virtual ("guest") operating system, and thus can install software and modify configuration settings as necessary. In addition, a virtual machine can be constrained to use specific amounts of computational resources (e.g., computer memory, processing power), thus enabling system administrators to ensure that multiple virtual machines can be executed simultaneously on the same computer without impacting each other's performance. After executing an analysis, the scientist can export the entire virtual machine to a single, binary file. Other scientists can then use this file to reconstitute the same computational environment that was used for the original analysis. With a few exceptions (see Discussion), these scientists will obtain exactly the same results as the original scientist. This process provides the added benefits that 1) the scientist must only document the installation and configuration steps for a single operating system, 2) other scientists need only install the virtualization software and not individual software components, and 3) analyses can be re-executed indefinitely, so long as the virtualization software remains compatible with current computer systems [86]. The fact that a team of scientists can employ virtual machines to ensure that each team member has the same computational environment is also useful because team members may have different configurations on their host operating systems.

One criticism of using virtual machines to support computational reproducibility is that virtual machine files are large (typically multiple gigabytes), especially if they include raw data files. This imposes a barrier for researchers to share virtual machines with the research community. One option is to use cloud-computing services (see Table 4). Scientists can execute an analysis in the cloud, take a "snapshot" of their virtual machine, and share it with others in that environment [84, 87]. Cloud-based services typically provide repositories where

Table 4 Commercial cloud-service providers

• Amazon Web Services [123]
• Rackspace.com/Cloud
• Google Cloud Platform [124]
• Windows Azure [125]

virtual machine files can easily be stored and shared among users. Despite these advantages, some researchers may prefer their data to reside on local computers, rather than in the cloud—at least while the research is being performed. In addition, cloud-based services may use proprietary software, so virtual machines may only be executable within each provider's infrastructure. Furthermore, to use a cloud service provider, scientists may need to activate a fee-based account.

When using virtual machines to support reproducibility, it is important that other scientists can not only re-execute the analysis, but also examine the scripts and code used within the virtual machine [88]. Although it is possible for others to examine the contents of a virtual machine directly, it is preferable to store the scripts and code in public repositories—separately from the virtual machine—so others can examine and extend the analysis more easily [89]. In addition, scientists can use a virtual machine that has been prepackaged for a particular research discipline. For example, CloudBioLinux contains a variety of bioinformatics tools commonly used by genomics researchers [90]. The scripts for building this virtual machine are stored in a public repository [91].

Scientists can automate the process of building and configuring virtual machines using tools such as Vagrant or Vortex (see Table 3). For either tool, users can write text-based configuration files that provide instructions for building virtual machines and allocating computational resources to them. In addition, these configuration files can be used to specify analysis steps [89]. Because these files are text based and relatively small (usually a few kilobytes), scientists can share them easily and track different versions of the files via source control repositories. This approach also mitigates problems that might arise during the analysis stage. For example, even when a computer's host operating system must be reinstalled because of a computer hardware failure, the virtual machine can be recreated with relative ease.

Software containers ease the process of installing and configuring dependencies

Software containers are a lighter weight alternative to virtual machines. Like virtual machines, containers encapsulate operating system components, scripts, code, and data into a single package that can be shared with others. Thus, as with virtual machines, analyses executed within a software container should produce identical outputs, irrespective of the underlying operating system or the software that may be installed outside the container (see Discussion for caveats). As is true for virtual machines, multiple containers can be executed simultaneously on a single computer, and each container may contain different software versions and configurations. However, whereas virtual machines include an entire

operating system, software containers interface directly with the computer's main operating system and extend it as needed (Fig. 6). This design provides less flexibility than virtual machines because containers are specific to a given type of operating system; however, containers require considerably less computational overhead than virtual machines, and can be initialized much more quickly [92].

The open source Docker.com utility, which has gained popularity among informaticians since its release in 2013, provides the ability to build, execute, and share software containers for Linux-based operating systems. Users specify a Docker container's contents using text-based commands. These instructions can be placed in a "Dockerfile", which other scientists can use to rebuild the container. As with virtual machine configuration files, Dockerfiles are text based, so they can be shared easily, and can be tracked and versioned in source control repositories. Once a Docker container has been built, its contents can be exported to a binary file; these files are generally smaller than virtual machine files, so they can be shared more easily—for example, via hub.Docker.com.

A key feature of Docker containers is that their contents can be stacked in distinct layers (or "images"). Each image includes software components to address a particular need. Within a given research lab, scientists might create general purpose images to support functionality for multiple projects, and specialized images to address the needs of specific projects. An advantage of Docker's modular design is that when images within a container are updated, Docker only needs to track the specific components that have changed; users who wish to update to a newer version must download a relatively small update. In contrast, even a minor change to a virtual machine would require users to export and reshare the entire virtual machine.

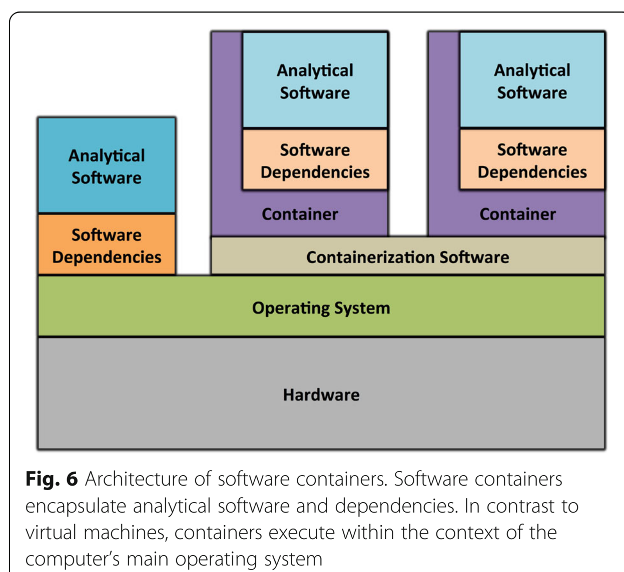

Fig. 6 Architecture of software containers. Software containers encapsulate analytical software and dependencies. In contrast to virtual machines, containers execute within the context of the computer's main operating system

Scientists have begun to share Docker images that enable others to execute analyses described in research papers [93–95], and to facilitate benchmarking efforts among researchers in a given subdiscipline. For example, nucleotid.es is a catalog of genome-assembly tools that have been encapsulated in Docker images [96, 97]. Genome assembly tools differ considerably in the dependencies they require, and in the parameters they support. This project provides a means to standardize these assemblers, circumvent the need to install dependencies for each tool, and perform benchmarks across the tools. Such projects may help to reduce the reproducibility burden on individual scientists.

The use of Docker containers for reproducible research comes with caveats. Individual containers are stored and executed in isolation from other containers on the same computer; however, because all containers on a given machine share the same operating system, this isolation is not as complete as it is with virtual machines. This means, for example, that a given container is not guaranteed to have access to a specific amount of computer memory or processing power—multiple containers may have to compete for these resources [92]. In addition, containers may be more vulnerable to security breaches [92]. Because Docker containers can only be executed on Linux-based operating systems, they must be executed within a virtual machine on Windows and Mac operating systems. Docker provides installation packages to facilitate this integration; however, the overhead of using a virtual machine offsets some of the performance benefits of using containers.

Efforts are ongoing to develop and refine software container technologies. Table 5 lists various tools that are currently available. In the coming years, these technologies promise to play an influential role within the scientific community.

Conclusions

Scientific advancement requires researchers to explicitly document the research steps they performed and to transparently share those steps with other researchers. This review provides a comprehensive, though not exhaustive, list of techniques that can help meet these requirements for computational analyses. Science philosopher Karl

Table 5 Open-source containerization software

- Docker.com
- LinuxContainers.org
- lmctfy [126]
- OpenVZ.org
- Warden [127]

Popper contended that, "[w]e do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them" [2]. Indeed, in many cases, the individuals who benefit most from computational reproducibility are those who performed the original analysis, but reproducible and transparent practices can also increase the level at which a scientist's work is accepted by other scientists [47, 98]. When other scientists can reproduce an analysis and determine exactly how its conclusions were drawn, they may be more apt to cite and build upon the work. In contrast, when others fail to reproduce research findings, it can derail scientific progress and may lead to embarrassment, accusations, and retractions.

We have described seven tools and techniques for facilitating computational reproducibility. None of these approaches is sufficient for every scenario in isolation; rather, scientists will often find value in combining approaches. For example, a researcher who uses a literate programming notebook (that combines narratives with code) might incorporate the notebook into a software container so that others can execute it without needing to install specific software dependencies. The container might also include a workflow management system to ease the process of integrating multiple tools and incorporating best practices for the analysis. This container could be packaged within a virtual machine or cloud-computing environment to ensure that it can be executed consistently (see Fig. 7). Binder [99] and Everware [100] are two services that allow researchers to execute Jupyter notebooks within a Web browser, using a Docker container to package the underlying software, and a cloud-computing environment to execute it. Although still under active development, such services may be harbingers of the future for computationally reproducible science.

The call for computational reproducibility relies on the premise that reproducible science will bolster the efficiency of the overall scientific enterprise [101]. Although reproducible practices may require additional time and effort, these practices provide ancillary benefits that help offset those expenditures [47]. Primarily, scientists may experience increased efficiency in their research [47]. For example, before and after a manuscript is submitted for publication, it faces scrutiny from co-authors and peer reviewers who may suggest alterations to the analysis. Having a complete record of all the analysis steps, and being able to retrace those steps precisely, makes it faster and easier to implement the requested alterations [47, 102]. Reproducible practices can also improve the efficiency of team science because colleagues can more easily communicate their research protocols and inspect each other's work; one type of relationship where this is critical is that between academic

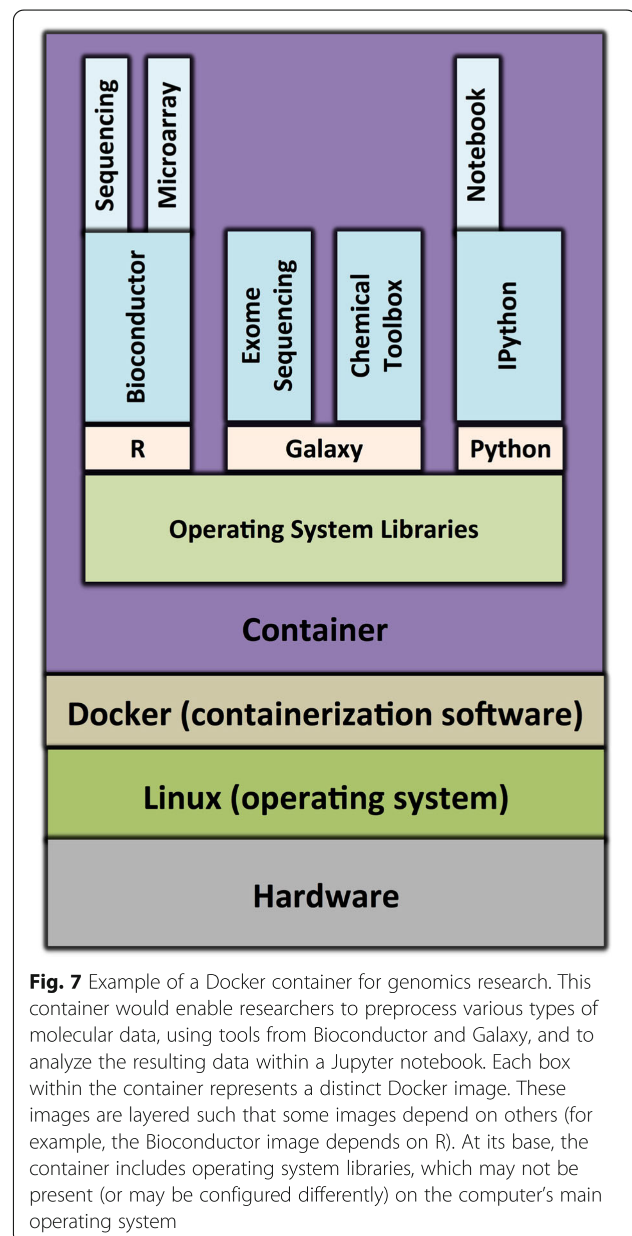

Fig. 7 Example of a Docker container for genomics research. This container would enable researchers to preprocess various types of molecular data, using tools from Bioconductor and Galaxy, and to analyze the resulting data within a Jupyter notebook. Each box within the container represents a distinct Docker image. These images are layered such that some images depend on others (for example, the Bioconductor image depends on R). At its base, the container includes operating system libraries, which may not be present (or may be configured differently) on the computer's main operating system

advisors and mentees [102]. Finally, when research protocols are shared transparently with the broader community, scientific advancement increases because scientists can learn more easily from each other's work and there is less duplication of effort [102].

Reproducible practices do not necessarily ensure that others can obtain identical results to those obtained by the original scientists. Indeed, this objective may be infeasible for some types of computational analysis, including those that use randomization procedures, floating-point operations, or specialized computer hardware [85, 103]. In such cases, the goal may shift to ensuring that others can obtain results that are semantically consistent with the original findings [5, 6]. In addition, in

studies where vast computational resources are needed to perform an analysis, or where data sets are distributed geographically [104–106], full reproducibility may be infeasible. Alternatively, it may be infeasible to reallocate computational resources for highly computationally intensive analyses [8]. In these cases, researchers can provide relatively simple examples to demonstrate the methodology [8]. When legal restrictions prevent researchers from publicly sharing software or data, or when software is available only via a Web interface, researchers should document the analysis steps as well as possible and describe why such components cannot be shared [25].

Computational reproducibility does not guarantee against analytical biases, or ensure that software produces scientifically valid results [107]. As with any research, a poor study design, confounding effects, or improper use of analytical software may plague even the most reproducible analyses [107, 108]. On one hand, increased transparency puts scientists at a greater risk that such problems will be exposed. On the other hand, scientists who are fully transparent about their scientific approach may be more likely to avoid such pitfalls, knowing that they will be more vulnerable to such criticisms. Either way, the scientific community benefits.

Lastly, we emphasize that some reproducibility is better than none. Although some of the practices described in this review require more technical expertise than others, they are freely accessible to all scientists, and provide long-term benefits to the researcher and to the scientific community. Indeed, as scientists act in good faith to perform these practices, where feasible, the pace of scientific progress will surely increase.

Open Peer Review

The Open Peer Review files are available for this manuscript as Additional files – See Additional file 5.

Additional files

Additional file 1: This script is supporting material for Fig. 1. It can be used to align DNA sequence data to a reference genome. First, it downloads the software and data files necessary for the analysis. Then, it extracts ("unzips") these files, and aligns the data to a reference genome for Ebola virus. Finally, it converts, sorts, and indexes the aligned data. (SH 865 bytes)

Additional file 2: This Make file is supporting material for Fig. 2. It performs the same function as Additional file 1, except that it is formatted for the Make utility. Accordingly, it is structured so that specific tasks must be executed before other tasks, in a hierarchical manner

Additional file 3: This Jupyter notebook is supporting material for Fig. 3. It contains code (in the Python programming language) for generating random numbers and plotting them in a graph. (IPYNB 53 kb)

Additional file 4: This document contains code (in the R language) for generating random numbers and plotting them on a graph. This document is in R Markdown format and can be compiled using knitr. (RMD 382 bytes)

Additional file 5: Open Peer Review. (PDF 6134 kb)

Abbreviations

DOI: Digital object identifier; HTML: Hypertext markup language; PDF: Portable document format; URL: Uniform resource locator; VCS: Version control system

Acknowledgements

SRP acknowledges startup funds provided by Brigham Young University. We thank research-community members and reviewers who provided valuable feedback on this manuscript.

Authors' contributions

SRP wrote the manuscript and created figures. MBF created figures and helped to revise the manuscript. Both authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Department of Biology, Brigham Young University, Provo, UT 84602, USA. [2]Department of Computer Science, Brigham Young University, Provo, UT, USA.

Published online: 11 July 2016

References

1. Fisher RA. The Design of Experiments. New York: Hafner Press; 1935.
2. Popper KR. The logic of scientific discovery. London: Routledge; 1959.
3. Peng RD. Reproducible research in computational science. Science. 2011;334:1226–7.
4. Russell JF. If a job is worth doing, it is worth doing twice. Nature. 2013;496:7.
5. Feynman RP. Six Easy Pieces: Essentials of Physics Explained by Its Most Brilliant Teacher. Boston, MA: Addison-Wesley; 1995. p. 34–5.
6. Murray-Rust P, Murray-Rust D. Reproducible Physical Science and the Declaratron. In: Stodden VC, Leisch F, Peng RD, editors. Implementing Reproducible Research. Boca Raton, FL: CRC Press; 2014. p. 113.
7. Hey AJG, Tansley S, Tolle KM, Others. The fourth paradigm: data-intensive scientific discovery. Redmond, WA: Microsoft Research Redmond, WA; 2009.
8. Millman KJ, Pérez F. Developing Open-Source Scientific Practice. Implementing Reproducible Research. Boca Raton, FL: CRC Press; 2014;149.
9. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. Best practices for scientific computing. PLoS Biol. 2014;12:e1001745.
10. Software with impact. Nat Methods. 2014;11:211.
11. Hong NC. We are the 92% [Internet]. Figshare; 2014. Available from: http://dx.doi.org/10.6084/M9.FIGSHARE.1243288. Accessed 1 March 2016.
12. Sacks J, Welch WJ, Mitchell TJ, Wynn HP. Design and analysis of computer experiments. Stat Sci. 1989;4:409–23.
13. Garijo D, Kinnings S, Xie L, Xie L, Zhang Y, Bourne PE, et al. Quantifying reproducibility in computational biology: the case of the tuberculosis drugome. PLoS One. 2013;8:e80278.
14. Error prone. Nature. 2012;487:406.
15. Vandewalle P, Barrenetxea G, Jovanovic I, Ridolfi A, Vetterli M. Experiences with reproducible research in various facets of signal processing research. IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP'07. IEEE. 2007;2007:IV-1253–6.
16. Cassey P, Cassey P, Blackburn T, Blackburn T. Reproducibility and repeatability in ecology. Bioscience. 2006;56:958–9.
17. Murphy JM, Sexton DMH, Barnett DN, Jones GS, Webb MJ, Collins M, et al. Quantification of modelling uncertainties in a large ensemble of climate change simulations. Nature. 2004;430:768–72.
18. McCarthy DJ, Humburg P, Kanapin A, Rivas MA, Gaulton K, Cazier J-B, et al. Choice of transcripts and software has a large effect on variant annotation. Genome Med. 2014;6:26.
19. Neuman JA, Isakov O, Shomron N. Analysis of insertion-deletion from deep-sequencing data: Software evaluation for optimal detection. Brief Bioinform. 2013;14:46–55.

20. Bradnam KR, Fass JN, Alexandrov A, Baranay P, Bechner M, Birol I, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. Gigascience. 2013;2:10.
21. Bilal E, Dutkowski J, Guinney J, Jang IS, Logsdon BA, Pandey G, et al. Improving breast cancer survival analysis through competition-based multidimensional modeling. PLoS Comput Biol. 2013;9:e1003047.
22. Gronenschild EHBM, Habets P, Jacobs HIL, Mengelers R, Rozendaal N, van Os J, et al. The effects of FreeSurfer version, workstation type, and Macintosh operating system version on anatomical volume and cortical thickness measurements. PLoS One. 2012;7:e38234.
23. Moskvin OV, McIlwain S, Ong IM. CAMDA 2014: Making sense of RNA-Seq data: From low-level processing to functional analysis. Systems Biomedicine. 2014;2:31–40.
24. Reducing our irreproducibility. Nature. 2013;496:398–398.
25. Michael CM, Nass SJ, Omenn GS, editors. Evolution of Translational Omics: Lessons Learned and the Path Forward. Washington, D.C: The National Academies Press; 2012.
26. Collins FS, Tabak L a. Policy: NIH plans to enhance reproducibility. Nature. 2014;505:612–3.
27. Chambers JM. S as a Programming Environment for Data Analysis and Graphics. Problem Solving Environments for Scientific Computing, Proceedings 17th Symposium on the Interface of Statistics and Computing North Holland; 1985. p. 211–4.
28. LeVeque RJ, Mitchell IM, Stodden V. Reproducible research for scientific computing: Tools and strategies for changing the culture. Comput Sci Eng. 2012;14:13.
29. Stodden V, Guo P, Ma Z. Toward reproducible computational research: an empirical analysis of data and code policy adoption by journals. PLoS One. 2013;8:2–9.
30. Morin A, Urban J, Adams PD, Foster I, Sali A, Baker D, et al. Research priorities. Shining light into black boxes. Science. 2012;336:159–60.
31. Rebooting review. Nat Biotechnol. 2015;33:319.
32. Ioannidis JP a, Allison DB, Ball C a, Coulibaly I, Cui X, Culhane AC, et al. Repeatability of published microarray gene expression analyses. Nat Genet. 2009;41:149–55.
33. Nekrutenko A, Taylor J. Next-generation sequencing data interpretation: enhancing reproducibility and accessibility. Nat Rev Genet. 2012;13:667–72.
34. Baggerly K a, Coombes KR. Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. Ann Appl Stat. 2009;3:1309–34.
35. Decullier E, Huot L, Samson G, Maisonneuve H. Visibility of retractions: a cross-sectional one-year study. BMC Res Notes. 2013;6:238.
36. Claerbout JF, Karrenbach M. Electronic Documents Give Reproducible Research a New Meaning. Meeting of the Society of Exploration Geophysics. New Orleans, LA; 1992.
37. Stodden V, Miguez S. Best practices for computational science: software infrastructure and environments for reproducible and extensible research. J Open Res Softw. 2014;2:21.
38. Ravel J, Wommack KE. All hail reproducibility in microbiome research. Microbiome. 2014;2:8.
39. Stodden V. 2014: What scientific idea is ready for retirement? [Internet]. http://edge.org/response-detail/25340. 2014. Available from: http://edge.org/response-detail/25340. Accessed 1 March 2016.
40. Birney E, Hudson TJ, Green ED, Gunter C, Eddy S, Rogers J, et al. Prepublication data sharing. Nature. 2009;461:168–70.
41. Hothorn T, Leisch F. Case studies in reproducibility. Brief Bioinform. 2011;12:288–300.
42. Schofield PN, Bubela T, Weaver T, Portilla L, Brown SD, Hancock JM, et al. Post-publication sharing of data and tools. Nature. 2009;461:171–3.
43. Piwowar H a., Day RS, Fridsma DB. Sharing detailed research data is associated with increased citation rate. PLoS One. 2007;2.
44. Johnson VE. Revised standards for statistical evidence. Proc Natl Acad Sci U S A. 2013;110:19313–7.
45. Halsey LG, Curran-everett D, Vowler SL, Drummond GB. The fickle P value generates irreproducible results. Nat Methods. 2015;12:179–85.
46. Wilson G. Software Carpentry: lessons learned. F1000Res. 2016;3:62.
47. Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten simple rules for reproducible computational research. PLoS Comput Biol. 2013;9:1–4.
48. GNU Make [Internet]. 2016. Available from https://www.gnu.org/software/make. Accessed 1 March 2016.
49. Make for Windows [Internet]. 2016. Available from http://gnuwin32.sourceforge.net/packages/make.htm. Accessed 1 March 2016.
50. Puppet [Internet]. 2016. Available from https://puppetlabs.com. Accessed 1 March 2016.
51. Code share. Nature. 2014;514:536.
52. Blischak JD, Davenport ER, Wilson G. A quick introduction to version control with Git and GitHub. PLoS Comput Biol. 2016;12:e1004668.
53. Loeliger J, McCullough M. Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. Sebastopol, California: "O'Reilly Media, Inc."; 2012. p. 456.
54. Huber W, Carey VJ, Gentleman R, Anders S, Carlson M, Carvalho BS, et al. Orchestrating high-throughput genomic analysis with Bioconductor. Nat Methods. 2015;12:115–21.
55. R Core Team. R: A Language and Environment for Statistical Computing [Internet]. Vienna, Austria: R Foundation for Statistical Computing; 2014. Available from: http://www.r-project.org. Accessed 1 March 2016.
56. Tóth G, Sokolov IV, Gombosi TI, Chesney DR, Clauer CR, De Zeeuw DL, et al. Space weather modeling framework: a new tool for the space science community. J Geophys Res. 2005;110:A12226.
57. Tan E, Choi E, Thoutireddy P, Gurnis M, Aivazis M. GeoFramework: Coupling multiple models of mantle convection within a computational framework. Geochem Geophys Geosyst. [Internet]. 2006;7. Available from: http://doi.wiley.com/10.1029/2005GC001155
58. Heisen B, Boukhelef D, Esenov S, Hauf S, Kozlova I, Maia L, et al. Karabo: An Integrated Software Framework Combining Control, Data Management, and Scientific Computing Tasks. 14th International Conference on Accelerator & Large Experimental Physics Control Systems, ICALEPCS2013. San Francisco, CA; 2013.
59. Schneider CA, Rasband WS, Eliceiri KW. NIH Image to ImageJ: 25 years of image analysis. Nat Methods. 2012;9:671–5.
60. Schindelin J, Arganda-Carreras I, Frise E, Kaynig V, Longair M, Pietzsch T, et al. Fiji: an open-source platform for biological-image analysis. Nat Methods. 2012;9:676–82.
61. Biasini M, Schmidt T, Bienert S, Mariani V, Studer G, Haas J, et al. OpenStructure: an integrated software framework for computational structural biology. Acta Crystallogr D Biol Crystallogr. 2013;69:701–9.
62. Ivy, the agile dependency manager [Internet]. 2016. Available from http://ant.apache.org/ivy. Accessed 1 March 2016.
63. aRchive: Enabling reproducibility of Bioconductor package versions (for Galaxy) [Internet]. 2016. Available from http://bioarchive.github.io. Accessed 1 March 2016.
64. Martin RC. Clean code: a handbook of agile software craftsmanship. Pearson Education. 2009.
65. Knuth DE. Literate programming. Comput J. 1984;27:97–111.
66. Pérez F, Granger BE. IPython: a system for interactive scientific computing. Comput Sci Eng. 2007;9:21–9.
67. Shen H. Interactive notebooks: Sharing the code. Nature. 2014;515:151–2.
68. Xie Y. Dynamic Documents with R and knitr. Boca Raton, FL: CRC Press; 2013. p. 216.
69. RStudio Team. RStudio: Integrated Development for R [Internet]. [cited 2015 Nov 20]. Available from: http://www.rstudio.com. Accessed 1 March 2016.
70. Gross AM, Orosco RK, Shen JP, Egloff AM, Carter H, Hofree M, et al. Multi-tiered genomic analysis of head and neck cancer ties TP53 mutation to 3p loss. Nat Genet. 2014;46:1–7.
71. Ding T, Schloss PD. Dynamics and associations of microbial community types across the human body. Nature. 2014;509:357–60.
72. Ram Y, Hadany L. The probability of improvement in Fisher's geometric model: A probabilistic approach. Theor Popul Biol. 2015;99:1–6.
73. Meadow JF, Altrichter AE, Kembel SW, Moriyama M, O'Connor TK, Womack AM, et al. Bacterial communities on classroom surfaces vary with human contact. Microbiome. 2014;2:7.
74. White E. Programming for Biologists [Internet]. Available from: http://www.programmingforbiologists.org. Accessed 1 March 2016.
75. Peng RD, Leek J, Caffo B. Coursera course: Exploratory Data Analysis [Internet]. Available from: https://www.coursera.org/learn/exploratory-data-analysis.
76. Bioconductor - Courses and Conferences [Internet]. [cited 2015 Nov 20]. Available from: http://master.bioconductor.org/help/course-materials. Accessed 1 March 2016.
77. Gil Y, Deelman E, Ellisman M, Fahringer T, Fox G, Gannon D, et al. Examining the challenges of scientific workflows. Computer. 2007;40:24–32.
78. Giardine B, Riemer C, Hardison RC, Burhans R, Elnitski L, Shah P, et al. Galaxy: a platform for interactive large-scale genome analysis. Genome Res. 2005;15:1451–5.

79. Goecks J, Nekrutenko A, Taylor J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. 2010;11:R86.
80. Afgan E, Baker D, Coraor N, Goto H, Paul IM, Makova KD, et al. Harnessing cloud computing with Galaxy Cloud. Nat Biotechnol. 2011;29:972–4.
81. Callahan SP, Freire J, Santos E, Scheidegger CE, Silva CT, Vo HT. VisTrails: Visualization Meets Data Management. Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM; 2006. p. 745–7.
82. Davidson SB, Freire J. Provenance and scientific workflows. Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD'08. 2008. p. 1345.
83. Lazarus R, Kaspi A, Ziemann M. Creating re-usable tools from scripts: The Galaxy Tool Factory. Bioinformatics. 2012;28:3139–40.
84. Dudley JT, Butte AJ. In silico research in the era of cloud computing. Nat Biotechnol. 2010;28:1181–5.
85. Hurley DG, Budden DM, Crampin EJ. Virtual Reference Environments: a simple way to make research reproducible. Brief Bioinform. 2015;16(5)901–903.
86. Gent IP. The Recomputation Manifesto. arXiv [Internet]. 2013; Available from: http://arxiv.org/abs/1304.3674. Accessed 1 March 2016.
87. Howe B. Virtual appliances, cloud computing, and reproducible research. Comput Sci Eng. 2012;14:36–41.
88. Brown CT. Virtual machines considered harmful for reproducibility [Internet]. 2012. Available from: http://ivory.idyll.org/blog/vms-considered-harmful.html. Accessed 1 March 2016.
89. Piccolo SR. Building portable analytical environments to improve sustainability of computational-analysis pipelines in the sciences [Internet]. 2014. Available from: http://dx.doi.org/10.6084/m9.figshare.1112571. Accessed 1 March 2016.
90. Krampis K, Booth T, Chapman B, Tiwari B, Bicak M, Field D, et al. Cloud BioLinux: pre-configured and on-demand bioinformatics computing for the genomics community. BMC Bioinformatics. 2012;13:42.
91. CloudBioLinux: configure virtual (or real) machines with tools for biological analyses [Internet]. 2016. Available from https://github.com/chapmanb/cloudbiolinux. Accessed 1 March 2016.
92. Felter W, Ferreira A, Rajamony R, Rubio J. An Updated Performance Comparison of Virtual Machines and Linux Containers [Internet]. IBM Research Division; 2014. Available from: http://domino.research.ibm.com/library/CyberDig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf. Accessed 1 March 2016.
93. Eglen SJ, Weeks M, Jessop M, Simonotto J, Jackson T, Sernagor E. A data repository and analysis framework for spontaneous neural activity recordings in developing retina. Gigascience. 2014;3:3.
94. Eglen SJ. Bivariate spatial point patterns in the retina: a reproducible review. Journal de la Société Française de Statistique. 2016;157:33–48.
95. Bremges A, Maus I, Belmann P, Eikmeyer F, Winkler A, Albersmeier A, et al. Deeply sequenced metagenome and metatranscriptome of a biogas-producing microbial community from an agricultural production-scale biogas plant. Gigascience. 2015;4:33.
96. Belmann P, Dröge J, Bremges A, McHardy AC, Sczyrba A, Barton MD. Bioboxes: standardised containers for interchangeable bioinformatics software. Gigascience. 2015;4:47.
97. Barton M. nucleotides · genome assembler benchmarking [Internet]. [cited 2015 Nov 20]. Available from: http://nucleotid.es. Accessed 1 March 2016.
98. Hones MJ. Reproducibility as a Methodological Imperative in Experimental Research. PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association. Philosophy of Science Association. 1990. p. 585–99.
99. Rosenberg DM, Horn CC. Neurophysiological analytics for all! Free open-source software tools for documenting, analyzing, visualizing, and sharing using electronic notebooks. J Neurophysiol American Physiological Society; Apr2016;jn.00137.2016.
100. everware [Internet]. 2016. Available from https://github.com/everware/everware. Accessed 1 March 2016.
101. Crick T. "Share and Enjoy": Publishing Useful and Usable Scientific Models. Available from: http://arxiv.org/abs/1409.0367v2. Accessed 1 March 2016.
102. Donoho DL. An invitation to reproducible computational research. Biostatistics. 2010;11:385–8.
103. Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Comput Surv. 1991;23:5–48.
104. Shirts M, Pande VS. COMPUTING: screen savers of the world unite! Science. 2000;290:1903–4.
105. Bird I. Computing for the large hadron Collider. Annu Rev Nucl Part Sci. 2011;61:99–118.
106. Anderson DP. BOINC: A System for Public Resource Computing and Storage. Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04). 2004.
107. Ransohoff DF. Bias as a threat to the validity of cancer molecular-marker research. Nat Rev Cancer. 2005;5:142–9.
108. Bild AH, Chang JT, Johnson WE, Piccolo SR. A field guide to genomics research. PLoS Biol. 2014;12:e1001744.
109. Köster J, Rahmann S. Snakemake—a scalable bioinformatics workflow engine. Bioinformatics. 2012;28:2520–2.
110. Sadedin SP, Pope B, Oshlack A. Bpipe : a tool for running and managing bioinformatics pipelines. Bioinformatics. 2012;28:1525–6.
111. Tange O. GNU Parallel - The Command-Line Power Tool.;login: The USENIX Magazine. Frederiksberg, Denmark; 2011;36:42–7
112. Albrecht M, Donnelly P, Bui P, Thain D. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. 2012.
113. Knight S, Austin C, Crain C, Leblanc S, Roach A. Scons software construction tool [Internet]. 2011. Available from: http://www.scons.org. Accessed 1 March 2016.
114. Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B, Mock S. Kepler: an extensible system for design and execution of scientific workflows. Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. IEEE; 2004. p. 423–4.
115. Goff SA, Vaughn M, McKay S, Lyons E, Stapleton AE, Gessler D, et al. The iPlant collaborative: cyberinfrastructure for plant biology. Front Plant Sci Frontiers. 2011;2:34.
116. Reich M, Liefeld T, Gould J, Lerner J, Tamayo P, Mesirov JP. GenePattern 2.0. Nat Genet. 2006;38:500–1.
117. Reich M, Liefeld J, Thorvaldsdottir H, Ocana M, Polk E, Jang D, et al. GenomeSpace: An environment for frictionless bioinformatics. Cancer Res. 2012;72:3966–3966.
118. GenePattern: A platform for reproducible bioinformatics [Internet]. 2016. Available from http://www.broadinstitute.org/cancer/software/genepattern]. Accessed 1 March 2016.
119. Wolstencroft K, Haines R, Fellows D, Williams A, Withers D, Owen S, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic Acids Res. 2013;41:557–61.
120. Rex DE, Ma JQ, Toga AW. The LONI pipeline processing environment. Neuroimage. 2003;19:1033–48.
121. LONI Pipeline Processing Environment [Internet]. 2016. Available from http://www.loni.usc.edu/Software/Pipeline. Accessed 1 March 2016.
122. Vortex [Internet]. 2016. Available from https://github.com/websecurify/node-vortex. Accessed 1 March 2016.
123. Amazon Web Services [Internet]. 2016. Available from http://aws.amazon.com. Accessed 1 March 2016.
124. Google Cloud Platform [Internet]. 2016. Available from https://cloud.google.com/compute. Accessed 1 March 2016.
125. Microsoft Azure [Internet]. 2016. Available from https://azure.microsoft.com. Accessed 1 March 2016.
126. lmctfy - Let Me Contain That For You [Internet]. 2016. Available from https://github.com/google/lmctfy. Accessed 1 March 2016.
127. Warden [Internet]. 2016. Available from http://docs.cloudfoundry.org/concepts/architecture/warden.html. Accessed 1 March 2016.