Dummy chapter needed for the \chapterauthor command to work later

1

Reproducible Bioinformatics Research for Biologists

Likit Preeyanon*

Microbiology & Molecular Genetics, Michigan State University, East Lansing, MI

Alexis Black Pyrkosz*

Avian Disease and Oncology Laboratory (ADOL), East Lansing, MI

C. Titus Brown

Computer Science & Engineering and Microbiology & Molecular Genetics, Michigan State University, East Lansing, MI

CONTENTS

[1]

1.1 Introduction

1.1.1 Computational analysis in pre-genomic era

At the dawn of computational biology in the 1960s, datasets were small. Protein sequences were first distributed in the printed Dayhoff atlases [29] and later on CD-ROM, with bioinformaticians eyeballing entire datasets and shuffling data by hand. By the 1990s, bioinformaticians were using spreadsheet programs and scientific software packages to analyze increasingly large datasets that included several phage and bacterial genomes. In 2003, the pre-genomic era ended with the online publication of the human genome[14, 7, 26] and the National Institutes of Health invested heavily in sequencing related organisms to aid in annotation. By the mid-2000s, Sanger sequencing was replaced by faster and cheaper next-generation sequencing technologies, resulting in an explosion of data, with bioinformaticians racing to develop automated and scalable computational tools to analyze and mine it[3].

1.1.2 Computational analysis in the era of next-generation sequencing

As sequencing becomes ever more affordable, the grand genomic and transcriptomic datasets that were the dream of many pre-genomic era biologists have become commonplace. A sin-

[1]*These authors contributed equally.

gle experiment in a small research lab can inform on thousands of genes or entire genomes, and small genomes can be sequenced and assembled in a few hours. Initiatives such as ENCODE[4, 19], 1000 Genomes Project[2], Cancer Genome Project[10], Human Microbiome Project[15], Eukaryotic Pathogen and Disease Vector Sequencing Project[20], Clinical Sequencing Exploratory Research[23], Centers for Mendelian Genomics[22], Environmental Genome Project[21], and HapMap Project[24] make vast amounts of data readily available for download and analysis. As the field comes closer to achieving the $1000 genome[16], waves of individual genomes will inundate the public databases, providing a rich information source for researchers to analyze with a wide array of tools. Further, proteomics, metabolomics, medical imaging, environmental conditions, and many other kinds of data are becoming readily available for integration. As scientists continue to push the edge of data analysis and integration, the integration of these different data types is increasingly required. The field has advanced far from the eye/hand methods of the pre-genomic era and outstripped the spreadsheets and single software packages of the early post-genomic era. Modern computational analyses are a major part of biological studies [32] and require analyzing gigabytes or terabytes of data in complex computational pipelines, which typically involve running several distinct programs with custom scripts porting data between them. These pipelines start from quality control of raw data (or by downloading primary data from public databases) and pass the data through many steps of calculation, validation, and statistics. They end with summarization and graphical visualization to aid end-users in comprehending the complex results. In short, modern biological studies require datasets that are so large, scientists must use advanced computational tools to perform useful analyses.

Genomics has expanded the drivers of science from hypothesis (devise a question and design/conduct experiments in response) to include discovery (sifting through large datasets in search of patterns). With this greater emphasis on statistical analyses of large datasets and data-driven modeling, even wet-lab biologists are increasingly finding themselves at the computer instead of the bench[31]. However, many biologists lack a strong background in mathematics or computer science[6, 27], and struggle to transition from a graphical computer desktop environment to the command line interface required for many analyses. Further, while they usually have been trained in good wet-lab practice, they often have minimal experience with computational practice and lack the knowledge necessary to efficiently perform high-quality reproducible computational research[31, 13]. Effectively, many biologists lack the computational skills they need to perform modern biological studies.

1.1.3 Concerns in Bioinformatic research

Biologists' lack of computational experience is a significant hindrance as biology expands to include data- and model-driven methodologies. While the obvious solution is to limit the computational aspect of biology to trained bioinformaticians and computer scientists, this is impractical for two reasons: not enough skilled bioinformaticians are available (only a minuscule percentage of US universities have bioinformatics undergraduate programs), and many computer scientists are uninterested/uneducated in science. Moreover, the substantial background in biology required to make appropriate use of data blocks computer scientists from quickly moving into bioinformatics. As a result, wet-lab biologists have begun to venture into computation in increasing numbers[8, 9]. They are usually long on data and short on time, so they focus on learning the computational tools needed to analyze their specific data, concentrating on rapidly processing data with the tools as opposed to understanding the tools' underlying assumptions. Even more troubling, there is a cultural and social gap because many labs and programs do not consider bioinformatics essential for their biologists[13]. A researcher seeking to analyze large datasets when few or none of his/her coworkers or superiors have computational expertise may have no clue where to begin, and be given very little time to find or develop appropriate tools. This situation fosters a dangerously ad hoc approach to bioinformatics.

The effects of this lack of expertise can be dire:

1. Many researchers download computational tools from the Internet or collaborators and use them on large datasets without first running a known test set (the computational equivalent of a control). Many programs contain technical or scientific errors that will be readily apparent when running test sets, but will be missed otherwise[9]. Errors will be carried into downstream analyses, costing hundreds of hours of compute and bench time, and potentially requiring retraction of papers when the errors are caught[31].
2. Many tools only run on the command line, are difficult to install, lack documentation, etc., and therefore software may be selected based on ease of use, rather than accuracy and scientific relevance.
3. With the trial and error approach used to create custom pipelines, biologists can lose track of which tools they ran, the order in which they ran them, and the parameter sets used for each. Many biologists have not carried the standard scientific practice of painstakingly recording wet-lab procedures in laboratory notebooks over into their computational research.
4. Many biologists use software with the default parameters. The defaults are frequently selected by the original programmers to optimize processing of the original test data or were based on a set of assumptions that was correct for the original study, but may not be appropriate for the different biologist's data or research question. While some parameters are relatively insensitive such that the defaults are sufficient, others will produce wildly different results if varied slightly. A single parameter can be the difference between one group's results being correct and another's being wrong.
5. Those biologists who program their own tools must decide how to release and support their code. Some labs post their software on a website but rarely update it. Frequently, as soon as a programmer leaves the lab, their code becomes unsupported and joins the online graveyard of dead and obsolete code. Some labs refuse to release their code[18], leading reviewers and collaborators to wonder if inaccuracies are being hidden.

The good news is that all of these problems can be addressed using tools and practices that are already available.

1.1.4 Reproducible research is attainable in Bioinformatics using modern tools

Many tools for reproducible computational research exist and are already being used in Computer Science, Physics, and Engineering. These tools are routinely used to quality control the data analysis process, facilitate useful collaborations, and maintain laboriously developed programs and pipelines in the long term. While these tools may be new to many biologists, they have been in production for many years and are well-tested with tutorials and online documentation. Investing time to learn the tools and establishing good habits of using them yields a larger benefit: errors are consistently detected and corrected early instead of being discovered only after time-consuming downstream analyses and attempted wet-lab verification (or after a paper has been submitted or published[17]). The more computational methods employed by a laboratory, the more essential the tools are to efficiency, correctness, and reproducibility [30]. Further, while use of these tools is currently optional in biology[12], researchers can expect that within a decade, most journals and granting agencies will require the appropriate use of tools and methods in computational biology research; the National Science Foundation already requires detailed discussion of data management.

1.1.5 Guidelines for getting started

Our goal is to help those biologists who have zero or little background in computation to get started with good practices and tools for computational science. The following sections are structured to provide introductory knowledge for those biologists venturing into bioinformatics and the command line interface for the first time; intermediate knowledge for those biologists ready to start programming; and advanced techniques for seasoned programmers for improving programs and automating pipelines; and a related tools section that names tools and concepts that readers can seek out once they have established a basic foundation for reproducible research.

Note: the following sections give overviews and simple examples of the tools, but readers are encouraged to use the resources listed at the end of the chapter to find specific information and step-by-step tutorials. Essentially, this chapter tells readers about existing tools and why they are important to use, but is not itself a course in computational research. If readers are interested in more hands-on experience with some of these topics, the Software Carpentry project (http://software-carpentry.org) offers free online videos as well as 2-day workshops in these and related areas.

1.2 Beginner

Here we discuss simple practices that beginners can use to establish a strong foundation for making their computational research reproducible, emphasizing those that are practical for scientists who primarily run other people's pipelines and are making the switch from a graphical user interface to the command line. We describe basic practices such as working on the command line and selecting a text editor.

1.2.1 Computing environment – the new benchtop

Just as much wet-lab biology work is done on a lab bench with routinely available tools such as micropipets, shakers, and spectroscopes, computational work is usually performed on a computer with routinely available data parsing and analysis tools. In this section, we will discuss computing environments (also sometimes called operating systems or platforms) and the general tools that stand ready on the computational benchtop.

1.2.1.1 UNIX/Linux operating system

Most biologists are aware of two primary computing platforms available: Windows and UNIX systems (which include Linux and Mac OS X). In the United States, most people learn basic computer skills on Windows machines. However, developers of bioinformatics software primarily use UNIX systems because of the large existing ecosystem of tools, most of which are open source – meaning that anyone can freely use, modify, and redistribute resources under open source licenses. The open source or free-software community has long attracted programmers and other technically-oriented people who creatively solve problems.

The Free Software Foundation[25] has numerous open source collections of development tools, libraries, licenses, and applications for the GNU/Linux system. Consequently, biologists can get an operating system and all the bioinformatics tools they need for free for all of their computers and clusters. Further, UNIX systems have traditionally been easier to use remotely than Windows machines, whether the machine is a desktop computer in another room, an institution's high performance compute cluster, or the cloud.

Making the switch from graphical user interface (GUI) based software to command line software is useful because the power to remix and combine tools at the command line outstrips that of most GUI-based software; as an analogy, GUI-based software is similar to a lab that relies exclusively on commercial kits whereas the command line is like a lab that is also equipped with chemicals and instruments that can be used to supplement the kits or develop novel techniques. Also, many GUI-based software packages require users to manually click through an analysis, while the command line can be used to write complete instructions for an analysis and run many datasets simultaneously. Automation ensures that each dataset is run according to the same instructions (avoiding human error) and unshackles the biologist from the computer. Some GUI and web-based programs are available for those biologists who want to build pipelines without using the command line (such as Taverna, Pipeline Pilot, and Galaxy). Further, most cutting-edge bioinformatics tools lack a GUI, partly because building GUIs is a time-consuming task that is difficult to fund. As a result, biologists need basic command line navigation skills to use the latest bioinformatics software. Biologists who use Windows but wish to take advantage of this cache of software and tools can download and install a free program such as Cygwin (or MSYS+Mingw32 or Microsoft Interix) to emulate a UNIX system on their Windows machine, or can use PowerShell.

Note: modern Macintosh machines are a good compromise for modern research labs because they have a UNIX-based system with a friendly GUI for casual use.

1.2.1.2 UNIX tools – the new benchtop tools

Just as wet-lab biologists use simple tools like pipets, centrifuges, incubators, and gel boxes individually and then combine them to perform a specific procedure, so bioinformaticians use simple UNIX tools that each perform a specific task and string them together into a pipeline. Many UNIX tools come pre-installed on UNIX systems or are freely available online. These tools are invaluable for the beginning bioinformatician, particularly if that researcher has no programming experience, because they will perform tasks with speed, customizability, and reliability that custom scripts cannot easily match. Here, we introduce some of the most basic, useful UNIX tools for bioinformatics.

Shell

The shell is a language as well as interpreter that reads and interprets commands from a user. Any biologist who has opened a terminal or command line interface and typed a command has used the shell. There are several types of shells available for UNIX, but bash is predominant. Bioinformaticians often use shell commands to run tools and automate tasks such as running pipelines, backing up data, and submitting jobs on a computer cluster. While shell languages can be used to develop full-fledged programs, this can be time-consuming because the shell is designed around operating system tasks rather than data analysis. Bioinformaticians usually use shell commands to perform routine tasks such as sorting large datasets, searching for specific data in a group of files, or sifting through a large log file and printing only the data relevant to a given project; they write programs to perform more complex tasks. Detailed below are some of the most general and useful tools on UNIX systems.

Grep, Sed, Cut, and Awk

grep, sed, cut, and *awk* are tools for parsing text files. *grep* is used to quickly search through a text file for a given word or sequence motif, similar to using a *find all* command in a word processor. *sed* is useful for replacing words or phrases, similar to using the *find and replace* command in a word processor. *cut* is used for selecting a column of data in a text file. *awk*, among many other uses, can search through files containing many columns of data and only print those lines or columns that are needed for a given application. Each tool is useful when the user needs to perform a single task quickly, create a simple pipeline to accomplish a combination of simple tasks, or process files that are too large to open in a spreadsheet program. *grep, sed,* and *awk* understand regular expression syntax (covered in the Related section), which offers more robust pattern searching options.

Apropos and Man

apropos is a program that displays a list of programs related to a keyword. It is useful for biologists who need to find a tool to perform a specific function without knowing the tool's name. For example, if a biologist wanted to archive files, he/she might use the *apropos* command to search for an appropriate tool:

```
$ apropos archive
```

Note: $ indicates the command prompt or where the user would begin typing. The user would not actually type the $.

The output will vary depending on system, but should look like:

```
jar(1) - Java archive tool
libtool(1) - create libraries ranlib - add or update the table of contents of
archive libraries
tar(1) - manipulate tape archives
unzip(1) - list, test and extract compressed files in a ZIP archive...
```

To learn more about each program, the biologist can look at the standard manual for each program using a *man* command (*man* is short for manual).

```
$ man tar
```

In this case, a man command will display a standard manual page, which typically includes the name of the program, synopsis, detailed description, and options as well as some examples. Here is an example of the first few lines of the standard manual for the tar program.

```
NAME
   tar -- manipulate tape archives

SYNOPSIS
   tar [bundled-flags <args>] [<file> | <pattern> ...] tar -c [options] [files
| directories] tar -r | -u -f archive-file [options] [files | directories] tar
-t | -x [options] [patterns]

DESCRIPTION
   tar creates and manipulates streaming archive files. This implementation can
extract from tar, pax, cpio, zip, jar, ar, and ISO 9660 cdrom images and can
```

```
create tar, pax, cpio, ar, and shar archives...
```

Note: Apropos and man commands serve as a reference, not a tutorial on how to use a particular command. Biologists may need to search in Google, Wikipedia, and other resources (such as `software-carpentry.org`*) to find tutorials, examples, and other information.*

History and Script

The shell automatically keeps a record of all commands used in a session. Typing *history* will print the list of commands. This tool is useful when a biologist is developing a computational procedure. Once the biologist has determined which commands and parameters are necessary to perform a required task, he/she can use the *history* tool to view and save the commands for future use (see next subsection).

If a biologist needs to save an interactive session at the command-line, he/she can use the *script* tool. A record will be generated for all data output to the terminal window.

1.2.1.3 Saving commands

One of the advantages of the command line is that biologists can save the exact commands and parameters used to perform a computational procedure, as opposed to a GUI-based procedure where it is difficult to record which buttons and options were used and the order in which they were clicked. At the most basic level, biologists can write the commands in their bound lab notebook. Another option is save the commands in a text file as a rudimentary electronic notebook so the biologist can search for a procedure later.

Example of shell commands to be written/typed in a bound or electronic notebook:

```
bowtie-build DataSet001.fa DataSet001.Index
bowtie -m 1 DataSet001.Index DataSet001Reads.fq DataSet001.map
```

In the example, the biologist is using software that aligns short reads to a reference sequence database. To use the software, two commands are required: (1) call the program bowtie-build to read the reference file *DataSet001.fa* and return the output files with the prefix *DataSet001.Index*, and (2) call the program bowtie with the parameter `-m 1` using the files from the previous step and the read file *DataSet001Reads.fq* as input and name the resulting alignment file *DataSet001.map*. By recording these commands exactly as typed in a notebook, the biologist will know the procedure used to generate the alignment file. If the biologist has more datasets, then these can be run using the same commands, changing only the file names. Further, when the biologist is writing up the results several months later, he/she can include the procedure so the results are credible and reproducible.

The most useful option is to create a short shell script. While this may seem daunting to a beginner, it is no more difficult than programming lab equipment (e.g. creating a PCR program on a thermocycler), and just as the PCR program is recorded in full in a notebook and used for all subsequent experiments, so can the shell script be painstakingly written and then simply used and referred to in later analyses.

Example bash script *alignMyRnaSeqData.sh*

```
#! /bin/bash
dataName=‘DataSet001’
params=‘-m 1’
bowtie-build $dataName‘.fa’ $dataName‘.Index’
bowtie $params $dataName‘.Index’ $dataName‘Reads.fq’ $dataName‘.map’
```

In the example bash script, the biologist has converted the previous example procedure into a series of commands with the name of the dataset and parameter list turned into variables. When the biologist needs to run it, he/she will type *bash alignMyRnaSeqData.sh* at the command prompt and the instructions will be executed automatically. The beginner can then manually edit the dataset name or parameter list with a text editor (see next subsection) each time it is run. Users who need to run the shell script on tens or thousands of files will benefit from learning a few extra commands so they can make the script loop through a list of file names. Using the script ensures that each time a dataset is run, the procedure remains the same, the output files are named systematically, and the biologist saves time by not having to retype commands or troubleshoot errors from typos.

Bash scripts are particularly useful when testing scientific software with different parameter sets. Biologists may not know that unlike laboratory commercial kits that have been rigorously tested on a variety of samples by experienced technicians, a significant number of scientific programs are written by graduate students and other academic researchers who are trying to solve a specific problem and optimize the parameters to that particular system. These default parameter sets are usually untested with other types of data unless the software has an active community of users who have found many of the problems. Therefore, biologists using new software should start by running a dataset with a known result (a control) on the default parameters, and then vary the parameters one by one to determine their sensitivity. (When possible, the parameters should be looked up in the documentation and through Internet searches to determine whether they are set to their optimal values for the current sample type.) Bash scripts are useful for testing parameter sets because biologists can set default parameters, run the script, change one parameter, and run the script again, confident that the only change to the procedure is the one they deliberately made. In the previous example, a biologist would vary the parameters listed in the params variable for each run. The results are more comparable and reproducible, and automating the procedure speeds up the parameter optimization process.

1.2.1.4 Text editors and IDEs – the bioinformaticians' word processors

Text editors are similar to word processors in that they are used to open, create/modify, and save text files and source code, but different because they do not save formatting characters or binary information in the file. Therefore, they produce the clean, simple files that are needed for running programs and analyzing data. There are many freely available editors with features such as graphical user interfaces, programming language-specific syntax highlighting, and advanced text parsing commands. We recommend that biologists learn cross-platform editors (i.e. can be used on Windows, UNIX systems, etc.), particularly those biologists who use Windows machines locally and UNIX systems remotely.

Two editors are particularly popular among computational scientists: Emacs and vi (or Vim). Both editors are cross-platform and have productivity-increasing features such as macros that can automate tasks, regular expressions to tailor/speed search and replace, etc. Emacs and vi can also be modified/customized using their internal scripting languages. This feature makes these two editors highly expandable and flexible. Numerous free plugins for both editors are developed and maintained by a large user-based community, which also provides free support for new users.

Integrated development environments (IDEs) provide tool sets including an editor, usually with advanced features, debugger, package manager, and numerous plugins. The most popular IDEs such as Eclipse and Netbeans support many languages including C++, Python, and Ruby, as well as HTML, PHP, and JavaScript for web development. IDEs also provide a nice GUI, which is built on top of command-line tools. Moreover, online tutorials are freely available for users of all levels. For biologists who intend to invest heavily in programming their own tools, IDEs may be a convenient step up from text editors.

1.3 Intermediate

This section addresses programming, the keystone of bioinformatics research. Biologists will be introduced to programming languages (and how to select one), good programming practices for developing less error-prone and more efficient code, program documentation for self and general use, version control systems as electronic notebooks and distribution methods, and controls for testing homegrown code. These sections are each intended as an overview of the tools and practices for writing scripts that are shorter than a page or two in length. References and tutorials for learning the languages and tools are included at the end of the chapter.

1.3.1 Programming

Programming is one of the most valuable skills for bioinformaticians. For example, just as an experienced wet-lab biologist might quickly pour a gel to purify a new sample, a bioinformatician will write a small script to filter a raw data file. Some scripts only contain a few lines of code, which are written for immediate use and then discarded. Therefore, most custom scripts are not documented, tested, or maintained. Major problems arise when hastily-written scripts are blindly reused for other projects or different datasets without proper quality control. In this section we discuss programming tools and approaches that are important for good computational lab practice.

1.3.2 Programming languages

Programming languages are sets of human-readable instructions that are translated into machine code to instruct a computer to perform a task. They are generally categorized as *interpreted* and *compiled*. Interpreted (scripting) languages (such as Python, Perl, R, and Ruby) use an interpreter program to run programs in one step, when the user is executing the program. Compiled languages (such as C, C++, and Java) use a two-step approach; the first step, compilation, runs during code development, and produces an executable file that contains only machine code. This file is then executed by the user. Both types of languages are widely used in bioinformatics. While compiled languages are generally more difficult to learn and time-consuming to use than their counterparts, the programs usually run faster because the code is translated in advance. This makes compiled languages more suitable for processing large datasets or performing complex or repetitive calculations. Interpreted languages are easier to learn and use, especially for beginners; for processing small datasets or relatively simple tasks, the difference in execution speed is negligible, while the advantage in development time can be significant.

Biologists with no programming experience should consider using a widely-used interpreted language such as Python or Perl. Both languages have a large user-based community, reference materials, and high-quality third-party libraries for a wide range of application domains. Another widely used, more domain-specific language for data analysis is R. R is a language of choice for many statisticians and bioinformaticians because of built-in data structures that are suitable for data analysis and a large number of libraries for complex statistical analyses and graphics. Biologists who work on high-throughput data analysis, such as microarrays and next-generation sequencing, may need to use libraries written in R or to run the analysis in R environments. Therefore, a knowledge of both R and Python or Perl languages is strongly encouraged.

While academia in particular has a culture of developing tools from scratch even when alternatives are available, this practice is increasingly challenging because pipelines are now too complex for a single novice programmer to develop quickly and accurately[11]. Third-party libraries are developed by programmers other than the developers of the language itself, and extend the language with custom written functions. For example, biologists who want to develop a bioinformatics web application can use Django, a Python third-party library that already contains most of the code required to build a Web application. The use of libraries is encouraged because they reduce errors in a program (i.e. a new programmer need not develop code that is susceptible to bugs when polished, well-tested code is already available). It also reduces program size, which increases maintainability (the programmer's ability to fix bugs and update code as upgrades to software and scientific methods become available). Many libraries are actively maintained, developed, and used by a community of programmers and scientists; therefore, they are well-tested and fairly reliable. Many libraries also support users via tutorials, online web forums, and mailing lists.

1.3.3 Good programming practices

Just as biologists follow standard lab practices in the wet lab, they should follow standard computational practices when writing code. For example, biologists frequently write short scripts to perform simple tasks. While writing a script, the meaning of each command and program logic is easy to understand. However, after a month or two, it can be nontrivial to determine the code function, inputs, etc. that were once so clear. Furthermore, as scripts change over time, multiple versions of each script may be scattered over multiple computers. This problem is akin to a biologist performing a quick procedure at the bench, and not taking the time to clean up or write down the procedure in detail.

In this section, we discuss basic programming practices that help programmers organize their code and make their scripts more reusable for other projects. We also introduce version control software, which facilitates code distribution among collaborators.

1.3.3.1 Code documentation

All code should be documented in plain English. The main purpose is to inform users about code function, expected input and output, and usage details, which are collectively called a code description. This is similar to laboratory equipment being packaged with standard operating procedures and troubleshooting guides that are available to biologists using the equipment. The programmer should remember that just as the wet-lab biologist will be more interested in using an instrument to do an experiment than opening the control panel and tracing circuits, so will users be focused on using a program to process data rather than reading source code, and should write the documentation accordingly.

Professional programmers conventionally write a code description at the beginning of each short program. Code descriptions should be short, providing maximum information in a minimum of words. For simple scripts, this may be two lines: one for the description and one to describe the usage. Some people might include the name of the author, date created, and date modified in the code description; however, when using a version control system (introduced later in this section), this practice is redundant.

When beginning programmers graduate to bundling scripts, programs, and libraries together, they should add a README file, which is a text file containing documentation for the entire code repository. The file usually includes the name of the author, contributors, installation, usage, and licenses for all programs in the repository.

Code readability is important. Experienced programmers customarily use two practices to improve readability: descriptive variable/function names and concise but clear comments. For example, a new programmer might write the following code and comment (comment shown in italics):

```
x += 1 # add 1 to x
```

Readers with minimal knowledge of Python will recognize that 1 is added to a variable *x*. The comment does not provide any information about the purpose of the statement or describe *x*. Instead, the code should be commented like this:

```
x += 1 # increase the count of DNA sequences read in
```

This way, a reader will quickly understand that *x* is the number of DNA sequences read and the number is being incremented by one. A more useful way to write this statement is:

```
sequence_count += 1 # increase the number of DNA sequences read by one
```

After choosing a more descriptive variable name, the comment is now largely redundant and could even be omitted. This is a simple example of how code readability can be improved by choosing appropriate variable names and commenting code sparingly.

1.3.3.2 Managing code/text with a version control system

A common change tracking nightmare is when a programmer creates a script and sends a copy to a collaborator, who we'll call Adam. After a period of time, Adam finds a bug in the first version of the script and informs the author. The author fixes the bug and sends Version 2 to Adam, not knowing that Adam previously sent Version 1 to other collaborators, Beth and Celia, who are not privy to Version 2 and therefore are likely to have unknowingly generated erroneous results with it. Further, Adam may inadvertently confuse Version 1 with Version 2 because the script name is the same for both and the only difference is in the code, which Adam cannot or does not read. Later, Beth (with Version 1) might try to compare results on a similar dataset with Adam's results from Version 2, and spend considerable time tracking down the reason for the differences. This would result in (at best) a re-discovery of the bug and (at worst) attempts to publish the erroneous comparison in a journal. The author, who has meanwhile upgraded to Version 5, is unaware of anything except that Adam was sent Version 2 a long time ago. This convoluted comedy of errors is one reason why a technique known as version control is essential in collaborations.

A version control system (VCS) is a program that tracks changes made to a file or set of files in a specified directory and records them on a central server. Users can add, remove, or edit files and the version control program will compare each file with the previous version and record the differences. For a single programmer, a VCS can be a rigorous, efficient electronic notebook of changes to a program: as the programmer creates/updates scripts, with a simple command, each change is meticulously recorded, dated, and archived. For a team of programmers, a VCS is a group notebook, distribution tool, and collaboration aid. Each programmer can access the latest version of the code and make changes. If multiple changes are made by different programmers to the same file(s), the changes are noted and automatically merged if possible. Conflicts (in which two programmers modify the same line or region of code) are flagged for manual resolution. For nonprogrammers, a VCS is a useful means of obtaining up-to-date software tools, as the VCS usually contains the most recently updated version of the software. Users can download code from the VCS and know exactly which version they are downloading.

Note: VCSs are also commonly used when collaborators are writing a paper using LaTeX or a similar document markup language. Each collaborator has immediate access to the most recent version of the paper, can make changes directly to the document, and commit them, as opposed to trading various versions via email or having one author decipher scribbled-on printouts.

This is particularly useful when a research lab has one programmer and multiple users because the programmer can create/edit scripts in the VCS and then users can check that VCS to determine if they need to download new/updated tools. When publishing computational analyses, study authors can note the version numbers of the scripts used so biologists wishing to reproduce the study can be given the correct version. Also, when the programmer leaves the lab, the latest versions of the tools are in the VCS so a newly hired replacement can immediately access and start maintaining the code.

Note: while setting up a VCS will require an initial investment of time, subsequent use is usually limited to a few simple commands. Some VCSs also include a GUI to help beginners and nonprogrammers use the system.

Some major version control systems (SVN, Git, Mercurial, etc.) are associated with websites that host repositories for free. For example, GitHub.com hosts more than 200,000 free code repositories for open source projects. These can be accessed freely from any part of the world. Using GitHub simplifies distribution because individual research labs do not need to constantly update lab websites with the latest version of a program; they can point lab members, collaborators, and blog readers to the online repository that the lab programmers are already updating. It is becoming common to publish a link to a GitHub repository to fulfill bioinformatic journals' requirement of open access software, which is advantageous because the site is separate from university or business websites that may change over time.

Note: biologists who prefer to keep their code for internal use only can set up Git or Mercurial to work locally only, pay Github for private repositories, or use Bitbucket. Alternatively, they can set up their own secure server (or have a network administrator set it up) and use Git or Subversion (SVN).

Real world example

In our lab, every programmer has a Github.com account for their projects. Some projects on Github are also linked to the lab repository `https://github.com/ged-lab`, which serves as the main repository for all source code and other materials written by lab members. We include a link to the main repository in each publication so anyone can download our source code and materials for using or reproducing results without first contacting us. In addition, anyone who finds a bug or wants to contribute to the project can do so by simply cloning the project, editing or adding code, and submitting a request to merge a change to the project. This opens up opportunities for improving the quality of scientific software as well as collaboration. This method has been proven quite successful in the open source community.

1.3.3.3 Basic code testing

Roughly one to ten programming errors occur per thousand lines of code[5, 12]. In this section, we discuss two techniques to help programmers find obvious bugs upfront: assert statements and doctests. We introduce more advanced tools such as unit tests and automatic testing systems in the *Related* section.

- Assert Statements

  The purpose of an assert statement is to compare a calculated value with an expected value and return true or false based on a programmer-defined condition. Assert statements can be used to test if code works as expected. They are particularly handy when testing *edge cases* such as when a user uses unexpected parameters or data files in unrecognized formats. For example, if we write a function *count_gc* in Python that returns the number of G and C nucleotides in a sequence, we could use assert statements to test the function:

```
assert count_gc("ATGTC") == 2
assert count_gc("ATTTTA") == 0
assert count_gc("") == None
```

  The first line tests whether the *count_gc* function correctly counts the number of G's and C's in the normal case or a mix of all four nucleotides. The second line tests if the function can correctly handle a calculation where there are no G's or C's present, which is also expected to be a common case. It is good programming practice to always test cases where zero is the expected result to ensure that it is correctly calculated and reported. The third line tests if the function recognizes that it has been passed an empty sequence and correctly reports an error; *None* is a defined value in Python that indicates "no result". In Python, a programmer can also specify an error message if an assert statement fails:

```
assert count_gc("") == None, "Empty sequence, should return None"
```

  In this case, if the function does not return *None*, the assert statement fails and prints "Empty sequence, must return None" on the computer screen. This error message alerts the programmer that the *count_gc* function is not handling the case correctly.

  Theoretically, assert statements should check all possible input values; however, this is not usually practical. In the above example, it would be impossible to generate every possible sequence that a user may input to the function. Therefore, a programmer will usually design a representative set of input data to systematically test the code. The previous example demonstrated this by testing both common and uncommon cases. The more assert statements added, the more likely an existing error will be found.

  Note: assert statements are also useful when the programmer is modifying the code later. Well-tested programs can be more easily modified and extended because the tests ensure that changes that break the existing code are more likely to be immediately discovered.

- Doctest

  Doctest is a useful feature in Python and several other languages that helps the programmer document and test his/her code simultaneously. This is particularly useful when writing documentation for developers planning to use or extend functions, because the doctests can ensure that the documentation examples are correct. Basically, doctest compares output from the Python interpreter with user-defined output. The test fails if they do not match. The doctest for the previous function would look like this:

```
>>>count_gc("ATGTC")
2
>>>count_gc("ATTTTA")
```

```
0
>>>count_gc("")
None
```

1.3.3.4 Code testing in real life

Effective testing catches errors. However, the human programmer can almost never consider and write tests for all possible ways of breaking code when he/she is developing the first version of a program; bugs are both inevitable and common. Therefore, writing tests should be incremental; each newly discovered bug should prompt the addition of a new test.

Tests are useful on many levels, but some programmers still do not write tests[11]. One reason is that test writing is not formally taught in most undergraduate computer science courses and therefore many programmers, let alone biologists, lack the required knowledge or experience. Another reason is because it is time-consuming and not considered a critical path activity[13, 9]. Rigorous tests may contain more lines of code than the actual code. However, there is some evidence that programmers who write tests spend less time debugging and produce higher quality code (see Chapter 12 of [1]). Moreover, time spent repeating an analysis because of a bug is usually far costlier than time spent writing tests.

1.3.4 A solid foundation

For many biologists, the guidelines introduced in the Beginner and Intermediate sections are sufficient to build a strong foundation for reproducible computational research. Beginners should be ready to investigate the tutorials and resources listed at the end of the chapter to build knowledge and experience with the command line interface: the new benchtop with open access, high-quality UNIX tools for data processing (without programming), simple bash scripts to generate reproducible procedures and optimize parameters, and text editors/IDEs to create and manipulate files across platforms. Intermediate users will be able to investigate various programming languages to find the one with the most high-quality libraries and support for their research area, document their code so it can be easily read and understood by other researchers, use a version control system as an electronic notebook and up-to-date distribution system for their evolving code, and write systematic tests to catch bugs early in the development and analysis process.

For most biologists, effectively using already developed software pipelines and writing small scripts to port data between them or processing large results files with UNIX tools is all they will need to complement their bench work. By incorporating these tools into their computational research and observing the computational lab safety practices, biologists can work effectively on the new benchtop, produce timely, accurate results that are simple to repeat with bash scripts or short, well-tested programs, and can distribute their programs per journal requirements using online version control so the research community can spend less time reinventing and fixing code and more time advancing science.

1.4 Advanced

Once a biologist has built a strong foundation for reproducible computational research, he/she may wish to progress to more complicated analyses, which accordingly require more complex calculations. The resulting programs can contain hundreds or thousands of lines of code, more instructions than a single human can keep in his/her head. An eager biologist who has been developing scripts with less than 50 lines may jump in and create a single file of several hundred lines. However, once a program has advanced beyond the simple script, new programming practices need to be followed to produce usable and maintainable code. This is analogous to chemists running a small reaction in the lab vs. chemical engineers scaling up a reaction to run in a chemical plant; process and resource management become significantly more important. As in the previous sections, these practices facilitate reproducibility, productivity, and frequently help maintain the programmer's sanity. In this section, we will discuss modularity (the practice of writing code in small blocks), refactoring/optimizing code performance for use with modern huge datasets, and using the IPython notebook as an interactive notebook/computing environment for integrating different programs and platforms and performing a complex analysis from start to finish.

Biologists/bioinformaticians grapple with a major problem before they start designing a new program: absence of detailed program specifications[13]. In the bioinformatics lab, programmers frequently have only a piece of the problem laid before them and minimal input from lab members. Bioinformaticians need to quickly develop a program that reads in data, performs a calculation, and writes results so lab members will react to the results, try to validate them, mention specifications/expectations that were not stated initially, or rethink the problem. The programmer is then expected to refine the program and show new results to solicit more feedback iteratively.

Productivity is the key to using this evolutionary approach to problem solving. Writing robust, reusable, and maintainable code is traded for writing code quickly because the programmer assumes that most code will be modified or discarded during the development process. Therefore, biologists/bioinformaticians should write code that is 1) functional, 2) readable, and 3) testable. Functionality is most important because if code does not work, then it is considered worthless[9]. Readability is necessary so the code can be understood by all programmers and users on the project. Testability is required because larger programs have more ways to break and therefore even more tests are needed.

1.4.1 Modularity

The first good practice for writing high-quality large programs is to divide code into small modules. A module is generally a small block of code that performs one specific task such as reading FASTA files, ensuring that a DNA sequence contains only the characters A, C, G, and T, or calculating the average of a set of numbers. The short scripts produced by intermediate biologists/bioinformaticians can easily be converted into modules; experienced software developers frequently write scripts that are simultaneously both. The purpose of creating modules is to take advantage of all the tools and practices discussed in the Intermediate section. It is also easier for a programmer to write logical and organized code when creating several small modules and linking them together than when writing one long linear program. These modules can then be bundled together to form a programmer's custom library. A large program ideally should consist of a main program file that accepts user input and then passes it through a series of modules or library functions.

There are several additional advantages to this practice:

1. Simplify and speed up programming. For example, a program may need to read several FASTA files to function. In a linear program, the code to load a file would need to be copied and pasted several times, which decreases the readability of the code. If the program is modular, the programmer need only create one module in the library and then reference it as many times as needed in the main program.
2. Library modules are easier to maintain. In a long, linear program that reads

multiple files, if a bug is found when reading the first FASTA file, then the error is likely to be in all instances of the code. Novice programmers frequently only fix those instances where an obvious error is shown, leaving silent errors in other parts of the program. In the modular example, the bug is fixed once in the module. Similarly, the module need only be tested once whereas good testing of a linear program would require many more tests.

3. Modules are easier to reuse for other projects. Once a programmer has written and tested a module and added it to his/her custom library, it can be used for all future projects. If the program is linear, the programmer must copy and paste lines of code from an old program to a new one, and then diligently test the code, or the programmer will need to reinvent the wheel by writing fresh code to load FASTA files for each new program.
4. The programmer can easily combine custom modules and third-party libraries to quickly and efficiently create large programs.
5. A team of programmers can easily collaborate on a large program when each of them is writing/testing different modules.

1.4.2 Code refactoring

Refactoring is the process of changing a program so the code is different but the results are same. This is similar to changing a wet-lab procedure so it uses fewer consumables, less dangerous chemicals, and less time while still generating a result of similar quality. Once a programmer has a well-tested program that produces the desired results, he/she can refactor the code so it is more readable and simpler, with emphasis on the readability. Before beginning, test codes should be written to ensure that the code still runs properly after refactoring and the code itself should be self-documenting (see Intermediate section).

When refactoring, there are several preferred practices:

1. Remove programming language-specific idioms or overly complex statements to increase readability. Many programming languages have similar structures and syntaxes (i.e. a Perl statement can be read by a C++ programmer as long as it does not use Perl-specific syntax). Also, overly complex statements tend to contain errors.
2. Divide large functions or modules into smaller ones. As discussed in the modularity section, smaller modules are easier to code, test, and reuse.
3. Remove dead/obsolete code. Because of the evolutionary process discussed previously, nonfunctional code from earlier versions might be lurking in the code. Removing obsolete lines will both increase readability and ensure that compute resources are not wasted on useless processing.

1.4.3 Code optimization

As mentioned previously, interpreted languages are often used to develop bioinformatics software because they reduce development time with the acceptable tradeoff that they are slower than compiled languages. In some cases, however, performance is critical and the program may be considered useless if it cannot achieve a particular speed. Optimization involves identifying the bottlenecks (the slowest sections) and modifying them to use different algorithms or compiled languages. This is similar to a wet-lab biochemist taking a multistep synthetic pathway, determining the rate-limiting steps, and either substituting those steps with new reactions or using catalysts. Generally, only the bottlenecks should be optimized; opening one or a few bottlenecks is usually sufficient to achieve the desired performance without spending time optimizing the entire program. The bottlenecks can be found using a tool called a profiler, which reports the time and the number of times a particular function or a method is called. Examples of a profiler are GNU gprof for C/C++ and profile, cProfile, and pstats module for Python.

Note: some common bottlenecks in bioinformatics programs involve loading data multiple times, unnecessary reading/writing data to disk, and inefficient searching using nested loops. These can often be solved by using more efficient algorithms or data structures. Novice programmers are prone to writing algorithms that mimic how a human would perform a task instead of harnessing the abilities of the computer and the specific programming language. Consultation with other bioinformaticians or with an online forum is a good path to improving code efficiency or solving specific problems.

Optimization depends on the languages being used. For a bioinformatics program written in Perl or Python, a bottleneck function can be rewritten in a compiled language like C++ or Fortran, and then wrapped so the interpreted language can use it. This method can increase performance by several magnitudes, although poorly written code in a compiled language may not function as well as well-written code in a scripting language. Experienced bioinformaticians utilize data structures and libraries that are built into the scripting and compiled languages, which have been optimized by professional software developers. For example, in Python, many built-in data structures and functions are actually implemented in C and wrapped so they can be called using Python code. A biologist/bioinformatician need only find an appropriate method that has already been optimized and implemented in a compiled language.

1.4.4 Research documentation

As stated previously, a complete bioinformatic analysis usually consists of running several third-party software packages, scripts that port data between them, and visualization tools to represent the final results. To reproduce the results, a biologist must repeat every step in the analysis in the correct order with the appropriate parameters. Until recently, providing a complete set of instructions was not trivial. Because the biologist/bioinformatician is frequently using an evolving procedure, the bookkeeping required for recording detailed procedures can become complicated. Fortunately, scientists and companies have developed tools to simplify the process, allowing researchers to conduct computational analyses while simultaneously building the final set of instructions. In this section, we introduce the IPython notebook, which has become popular due to its support of Python, shell commands, and R (a statistics and graphing language).

1.4.4.1 IPython notebook

IPython notebook is a combined electronic notebook and programming/computing environment. Users can create a notebook for a project and link all scripts, programs, and shell commands, and parameters used to the notebook, including the order in which they are to be run. Users can then run the analysis from start to finish in the notebook and view/save output at each step as well as add textual notes and comments. This high level of organization can boost biologists/bioinformaticians' productivity while giving them the tools to run and rerun their analyses reproducibly. A single click will run an entire pipeline, expediting parameter optimization, replicate runs, and reruns after fixing bugs. Automating the process in the notebook minimizes mistakes from typos and other human errors. Moreover, IPython notebook can be run on a remote server, making it suitable for computer clusters or cloud computing systems. The notebook can be distributed to collaborators who can rerun all commands and see identical results on their computer without the programmer writing any additional documentation.

The notebook is not efficient for running processes that require days to finish. Therefore, we write shell scripts to perform the laborious number-crunching and use the notebook for everything else.

The IPython notebook is built on top of IPython, an advanced Python shell for interactive Python programming, but can support many more programming languages. For example, with the rmagic plugin and Python RPy2 library, users can use R libraries such as those from Bioconductor (see *Resources*) and store the results in Python data structures, which can be further analyzed with Python. This feature reduces the number of steps in data transformation, which is a common problem in bioinformatics research. Another useful feature of IPython notebook is an in-line plot, which allows users to use Matplotlib library or R to make and visualize a plot, and then save it in IPython notebook with comments and code to create an executable document. Finally, the notebook is stored as a plain text file that can be version controlled and distributed as discussed above. IPython has many more plugins and shortcuts that support scientific computing analyses. Shown below is an example of boosting productivity by using a shortcut:

```
>>> expression_values =!cut -f 2 expression.dat # read in a value from the
second column of a text file
```

Expression values are contained in expression.dat, and *cut* is used to select the second data column (*-f 2*). This data is then assigned to the Python list variable *expression_values* for later use. Running this single line in the notebook can take the place of running the *cut* command at the command line, saving the results to a file, opening that file in Python, reading the data, and then assigning it to the list variable. Throughout this chapter we have emphasized the idea that biologists/bioinformaticians can benefit greatly by utilizing existing well-tested tools as opposed to reinventing the wheel. IPython has been used by many scientists for several years; therefore, it is not surprising that there are many commands, shortcuts, and plugins that perform common tasks elegantly, accurately, and expeditiously. IPython also allows users to create their own plugins to extend its functionality.

Note: Similar "notebook"-like tools exist for R, as well.

Real World Example

The senior author (C. Titus Brown, MSU) used the IPython notebook to complete parts of a bioinformatics analysis for a recent manuscript, available at `https://github.com/ged-lab/2012-paper-diginorm.git`. The author also provides a tutorial on running the pipeline and reproducing the results using IPython notebook at `http://ged.msu.edu/angus/diginorm-2012/pipeline-notes.html`. The analysis was tested on Amazon cloud service with *ami-61885608*, which has all the required programs pre-installed. Anyone can follow the pipeline and use the notebook to reproduce the analysis from start to finish with identical results with minimal effort.

At the NIH Cloud Computing for the Microbiome workshop in 2012, a team of researchers from different backgrounds used IPython notebook and StarCluster as collaboration tools to produce publishable results in record time [28].

1.5 Related Topics

For biologists/bioinformaticians who have progressed through the Beginner to Advanced sections, we briefly describe more advanced computing topics that can facilitate accurate, efficient computational research.

1.5.1 Using online resources

A wide range of topics relevant to programming, bioinformatics, and data analysis are discussed online in blogs, Web forums, and Twitter. Perhaps the single most useful approach to problem solving available is to do an online search for your problem; if it is a problem or bug that has been encountered in a popular piece of software, a solution will almost certainly have been posted.

Two particularly useful Web forums for bioinformatics are BioStars and Seqanswers. We strongly suggest that novice bioinformaticians search these forums for discussions of tools. Both of these forums also support asking questions, and the online bioinformatics community is generally very friendly and willing to help; we encourage you to first search to see if someone has already asked your question, and if not, to then post the question on one or both of these forums.

1.5.2 Advanced tools

1.5.2.1 Regular expressions

Regular expressions are tools for searching text for a particular pattern of letters/numbers/symbols. For example, a bioinformatician can search for short DNA sequence motifs in a data file with an unstructured or unusual format. Regular expressions have their own syntax for defining a specific pattern, which to the casual eye can look like an unintuitive shell language. For example, `logy\b` defines a pattern for a word that ends with *logy*, matching biology, physiology, technology, etc. In programming, regular expressions are used to concisely and quickly search for patterns in data.

1.5.2.2 Debuggers

Code frequently contains errors or bugs that cause unexpected results. While syntax errors will be caught by the interpreter or compiler, logic errors often go undetected. A debugger allows programmers to interact with their program by running and pausing execution, stepping in and out of functions or loops, and changing values in variables during execution to locate a bug. Most programming languages have at least one debugger. GDB is a standard GNU/Linux debugger that can be used with most compiled programming languages. Moreover, major IDEs such as Eclipse and Netbeans have built-in debuggers with GUIs.

1.5.2.3 Unit tests and automated testing

Unit testing consists of writing code that tests individual units of a program, such as functions or modules. Each subunit is tested in isolation; therefore, tests on a given subunit will not be affected by bugs from other subunits. This procedure helps locate errors in a large program. Unit testing libraries are available for most major languages and help users create test suites. Some libraries also provide a test runner to run tests automatically. An important advantage of automatic testing is that users can test the program to ensure that the installation process is bug-free. Unit testing also promotes a test-driven development process, which helps guarantee that every function works as expected and tests are written for every function in a program.

Real world example

In our lab, large programming projects have separate folders for test code. However, each project uses different libraries for testing; for example, Gimme (`https://github.com/likit/gimme`) uses a Python unittest module whereas khmer (`http://github.com/ged-lab/khmer.git`) uses user-defined functions, which are recognized and run by the nosetest module. Instructions for automatically running tests are included in the corresponding README files.

1.5.3 Advanced programming topics

1.5.3.1 Object-oriented programming paradigm

Object-Oriented Programming (OOP) is a standard programming paradigm that creates "objects," which usually consist of data and specific methods for operating on that data. This is useful when a large program needs to have standardized methods for data-processing but also uses several different types of data. In addition, objects can be reused or extended without rewriting them from scratch. This practice has been extensively used throughout the software industry and in scientific programming because it promotes code maintenance and expansion.

1.5.3.2 Algorithms and data structures

Most major languages used in scientific computing provide libraries supporting well-optimized algorithms and data structures. However, using a pre-implemented algorithm without understanding the underlying concepts is unsafe. Most algorithms have strengths and weaknesses that should be evaluated based on the specific application. Basic knowledge of algorithms will help biologists make a correct decision.

1.5.3.3 Compiled languages

While time-consuming, learning a compiled language such as C/C++ gives insight into how a computer functions because it requires machine-based knowledge. Scripting languages are designed to abstract away many low-level details to improve programmer productivity. As a result, it also abstracts away important concepts of computing such as memory management. A basic understanding of how to program in a compiled language will help biologists write better code using scripting languages because of a greater understanding of the underlying mechanisms. In addition, many bioinformatics software packages still use code in compiled languages to do rapid processing.

1.6 Conclusion

The general problem in the bioinformatics field is not an absence of tools and good practices, but rather that many researchers lack knowledge and training with them[31]. Embedded in a scientific culture that is relatively inexperienced with good computational practices, many biologists make their first foray into bioinformatics with only intuition and the Internet to guide them. The tools and practices discussed here are intended to help those biologists build a solid foundation in reproducible computational research. Because it would be impossible to condense several years of scientific computing and data analysis training into a single book chapter, we have focused on describing those tools and practices that are particularly useful for novice bioinformaticians, emphasizing their contribution to productivity and reproducibility, with the intention of giving biologists the introduction they need to then seek specific information and step-by-step tutorials elsewhere.

Investing in learning computational tools and practices will yield incalculable return over the course of a biologist's career. With the enormous potential for discovery available in this era of Big Data, datasets can be expected to continue expanding. A biologist who starts learning and applying general tools and good computing practices now will eventually save *years* of time. Moreover, the biologist who invests time in writing tests, using version control, and automating/distributing analyses in the IPython notebook will both be compliant with journal and granting agency policies for distributing code and avoid the aggravation of other scientists repudiating his/her unreproducible results.

While the list of tools and practices to learn might seem overwhelming to biologists with no prior computational experience, we encourage them to take a systematic approach to the education process. Just as a wet-lab researcher is trained on one instrument at a time, practices using it for his/her current project, and then moves on to more complicated methods, so too can he/she learn computational techniques. Installing software for running a command line interface and looking through a text file of data is a good start. This simple task can build confidence with using the command line, and soon the biologist will be ready to learn simple *grep* commands to make looking through that text file easier and faster. Each skill will build upon the last, and the biologist will soon be applying these skills to his/her research, finding experiments where a new option or tool will accurately process data in seconds that would otherwise have required hours of mind-numbing clicking. Once the biologist gains experience with the Beginner tools and practices, he/she will be ready to tackle the Intermediate section, and the tools and practices described will seem like a natural progression. As the biologist becomes a more practiced bioinformatician with a well-stocked toolbox, so too will his/her research advance.

A single biologist using good computational tools and practices can produce a lifetime of biological breakthroughs and innovation. A team of skilled biologists can work in parallel to push the limits of biological knowledge in a particular area by several lifetimes. As more biologists practice reproducible computational research, the sheer breadth and depth of their work will collectively move the entire field of biology into a new era of scientific discovery.

1.7 Acknowledgments

The authors thank Greg Wilson, Steve Haddock, Hans Cheng, Randy Olson, Eric McDonald, Jason Pell, and Cari Hearn for reviewing early drafts of this manuscript.

This chapter is maintained online at http://reproducibility.idyll.org/, where it is being continuously modified and updated by the authors and members of the scientific community. Readers are encouraged to visit the associated forum and leave questions/comments.

1.8 Available Resources

1.8.1 Books

1.8.1.1 UNIX/Linux tools

- Haddock & Dunn *Practical Computing for Biologists*
- Newham, Cameron *Learning the bash Shell* [O'Reilly]
- Robbins, Arnold and Dougherty, Dale *sed & awk* [O'Reilly]
- Cameron, Debra et al. *Learning GNU Emacs* [O'Reilly]
- Robbins, Arnold et al. *Learning the vi and Vim Editors* [O'Reilly]
- Neil, Drew *Practical Vim: Edit Text at the Speed of Thought* [Pragmatic Bookshelf]
- Chacon, Scott *Pro Git* [Apress]

1.8.1.2 Python

- Campbell, Gries, Montojo and Wilson *An introduction to computer science using Python* [Pragmatic Bookshelf]
- Lutz, Mark *Learning Python* [O'Reilly]
- Model, L Mitchell *Bioinformatics Programming Using Python* [O'Reilly]
- Vaingast, Shai *Beginning Python Visualization* [Apress]
- Arbuckle, Daniel *Python Testing: Beginner's Guide* [Packtpub]

1.8.1.3 Others

- Joe Pitt-Francis, Jonathan Whiteley *Guide to Scientific Computing in C++* [Springer]
- James Tisdall *Beginning Perl for Bioinformatics* [O'Reilly]
- James Tisdall *Mastering Perl for Bioinformatics* [O'Reilly]
- Ellie Quigley *Perl by Example* [Prentice Hall]
- Peter Cooper *Beginning Ruby* [Apress]
- Joseph Adler *R in a Nutshell* [O'Reilly]
- Peter Dalgaard *Introductory Statistics with R* [Springer]
- Paul Teetor *R Cookbook* [O'Reilly]

1.8.2 Online resources

1.8.2.1 UNIX/Linux Tools

- GNU Operating System
  `http://www.gnu.org`
- Cygwin (Linux emulator for Windows)
  `http://www.cygwin.com`
- MSYS+MinGW
  texttthttp://www.mingw.org/wiki/MSYS
- Vi and Vim
  `http://www.vim.org/index.php`
- Emacs
  `http://www.gnu.org/software/emacs/`
- Github: Git online repository
  `http://github.com`
- Git tutorial
  `http://git-scm.com`
- Mercurial
  `http://mercurial.selenic.com/`
- SVN
  `http://subversion.apache.org`

1.8.2.2 Python

- Python: Python official website
  `http://python.org`
- Python style guide
  `http://www.python.org/dev/peps/pep-0008/`
- The Zen of Python: A guideline for Python coding
  `http://www.python.org/dev/peps/pep-0020/`
- Learn programming by visualizing code execution
  `http://www.pythontutor.com/`
- Python doctests
  `http://docs.python.org/library/doctest.html`
- Python unittest
  `http://docs.python.org/library/unittest.html`
- IPython: Advanced Python shell
  `http://ipython.org`
- Scipy: Scientific tools for Python
  `http://www.scipy.org/`
- Matplotlib: Python plotting library
  `http://matplotlib.org/`

- Python: Speed and Performance tips
  `http://wiki.python.org/moin/PythonSpeed/PerformanceTips`
- Performance analysis of Python programs
  `http://www.doughellmann.com/PyMOTW/profile/`

1.8.2.3 R

- R Official website:
  `http://www.r-project.org/`
- Rseek: Search engine for R related materials
  `http://rseek.org`
- Bioconductors: R packages for bioinformatics
  `http://bioconductor.org`

1.8.2.4 Web Forums

- BioStars: Bioinformatics answers
  `http://www.biostars.org/`
- Seqanswers: Bioinformatics answers
  `http://seqanswers.com`
- Stack Overflow: General programming
  `http://stackoverflow.com/`

1.8.2.5 Others

- Software carpentry: Online training
  `http://software-carpentry.org`
- Rosalind: Learning Bioinformatics
  `http://rosalind.info/problems/as-table/`
- Reproducible Research
  `http://reproducibleresearch.net`
- Analyzing Next-Generation Sequencing Data
  `http://bioinformatics.msu.edu/ngs-summer-course-2012`

Bibliography

[1] Oram A. and Wilson G.V., editors. *Making Software: What Really Works, and Why We Believe It.* Farnham: O'Reilly, 2011.

[2] D. Altshuler, R. M. Durbin, G. R. Abecasis, D. R. Bentley, A. Chakravarti, A. G. Clark, F. S. Collins, F. M. De La Vega, P. Donnelly, M. Egholm, P. Flicek, S. B. Gabriel, R. A. Gibbs, B. M. Knoppers, E. S. Lander, H. Lehrach, E. R. Mardis, G. A. McVean, D. A. Nickerson, L. Peltonen, A. J. Schafer, S. T. Sherry, J. Wang, R. Wilson, R. A. Gibbs, D. Deiros, M. Metzker, D. Muzny, J. Reid, D. Wheeler, J. Wang, J. Li, M. Jian, G. Li, R. Li, H. Liang, G. Tian, B. Wang, J. Wang, W. Wang, H. Yang, X. Zhang, H. Zheng, E. S. Lander, D. L. Altshuler, L. Ambrogio, T. Bloom, K. Cibulskis, T. J. Fennell, S. B. Gabriel, D. B. Jaffe, E. Shefler, C. L. Sougnez, D. R. Bentley, N. Gormley, S. Humphray, Z. Kingsbury, P. Kokko-Gonzales, J. Stone, K. J. McKernan, G. L. Costa, J. K. Ichikawa, C. C. Lee, R. Sudbrak, H. Lehrach, T. A. Borodina, A. Dahl, A. N. Davydov, P. Marquardt, F. Mertes, W. Nietfeld, P. Rosenstiel, S. Schreiber, A. V. Soldatov, B. Timmermann, M. Tolzmann, M. Egholm, J. Affourtit, D. Ashworth, S. Attiya, M. Bachorski, E. Buglione, A. Burke, A. Caprio, C. Celone, S. Clark, D. Conners, B. Desany, L. Gu, L. Guccione, K. Kao, J. Kebbler, J. Knowlton, M. Labrecque, L. McDade, C. Mealmaker, M. Minderman, A. Nawrocki, F. Niazi, K. Pareja, R. Ramenani, D. Riches, W. Song, C. Turcotte, S. Wang, E. R. Mardis, R. K. Wilson, D. Dooling, L. Fulton, R. Fulton, G. Weinstock, R. M. Durbin, J. Burton, D. M. Carter, C. Churcher, A. Coffey, A. Cox, A. Palotie, M. Quail, T. Skelly, J. Stalker, H. P. Swerdlow, D. Turner, A. De Witte, S. Giles, R. A. Gibbs, D. Wheeler, M. Bainbridge, D. Challis, A. Sabo, F. Yu, J. Yu, J. Wang, X. Fang, X. Guo, R. Li, Y. Li, R. Luo, S. Tai, H. Wu, H. Zheng, X. Zheng, Y. Zhou, G. Li, J. Wang, H. Yang, G. T. Marth, E. P. Garrison, W. Huang, A. Indap, D. Kural, W. P. Lee, W. F. Leong, A. R. Quinlan, C. Stewart, M. P. Stromberg, A. N. Ward, J. Wu, C. Lee, R. E. Mills, X. Shi, M. J. Daly, M. A. DePristo, D. L. Altshuler, A. D. Ball, E. Banks, T. Bloom, B. L. Browning, K. Cibulskis, T. J. Fennell, K. V. Garimella, S. R. Grossman, R. E. Handsaker, M. Hanna, C. Hartl, D. B. Jaffe, A. M. Kernytsky, J. M. Korn, H. Li, J. R. Maguire, S. A. McCarroll, A. McKenna, J. C. Nemesh, A. A. Philippakis, R. E. Poplin, A. Price, M. A. Rivas, P. C. Sabeti, S. F. Schaffner, E. Shefler, I. A. Shlyakhter, D. N. Cooper, E. V. Ball, M. Mort, A. D. Phillips, P. D. Stenson, J. Sebat, V. Makarov, K. Ye, S. C. Yoon, C. D. Bustamante, A. G. Clark, A. Boyko, J. Degenhardt, S. Gravel, R. N. Gutenkunst, M. Kaganovich, A. Keinan, P. Lacroute, X. Ma, A. Reynolds, L. Clarke, P. Flicek, F. Cunningham, J. Herrero, S. Keenen, E. Kulesha, R. Leinonen, W. M. McLaren, R. Radhakrishnan, R. E. Smith, V. Zalunin, X. Zheng-Bradley, J. O. Korbel, A. M. Stutz, S. Humphray, M. Bauer, R. K. Cheetham, T. Cox, M. Eberle, T. James, S. Kahn, L. Murray, A. Chakravarti, K. Ye, F. M. De La Vega, Y. Fu, F. C. Hyland, J. M. Manning, S. F. McLaughlin, H. E. Peckham, O. Sakarya, Y. A. Sun, E. F. Tsung, M. A. Batzer, M. K. Konkel, J. A. Walker, R. Sudbrak, M. W. Albrecht, V. S. Amstislavskiy, R. Herwig, D. V. Parkhomchuk, S. T. Sherry, R. Agarwala, H. M. Khouri, A. O. Morgulis, J. E. Paschall, L. D. Phan, K. E. Rotmistrovsky, R. D. Sanders, M. F. Shumway, C. Xiao, G. A. McVean, A. Auton, Z. Iqbal, G. Lunter, J. L.

Marchini, L. Moutsianas, S. Myers, A. Tumian, B. Desany, J. Knight, R. Winer, D. W. Craig, S. M. Beckstrom-Sternberg, A. Christoforides, A. A. Kurdoglu, J. V. Pearson, S. A. Sinari, W. D. Tembe, D. Haussler, A. S. Hinrichs, S. J. Katzman, A. Kern, R. M. Kuhn, M. Przeworski, R. D. Hernandez, B. Howie, J. L. Kelley, S. C. Melton, G. R. Abecasis, Y. Li, P. Anderson, T. Blackwell, W. Chen, W. O. Cookson, J. Ding, H. M. Kang, M. Lathrop, L. Liang, M. F. Moffatt, P. Scheet, C. Sidore, M. Snyder, X. Zhan, S. Zollner, P. Awadalla, F. Casals, Y. Idaghdour, J. Keebler, E. A. Stone, M. Zilversmit, L. Jorde, J. Xing, E. E. Eichler, G. Aksay, C. Alkan, I. Hajirasouliha, F. Hormozdiari, J. M. Kidd, S. C. Sahinalp, P. H. Sudmant, E. R. Mardis, K. Chen, A. Chinwalla, L. Ding, D. C. Koboldt, M. D. McLellan, D. Dooling, G. Weinstock, J. W. Wallis, M. C. Wendl, Q. Zhang, R. M. Durbin, C. A. Albers, Q. Ayub, S. Balasubramaniam, J. C. Barrett, D. M. Carter, Y. Chen, D. F. Conrad, P. Danecek, E. T. Dermitzakis, M. Hu, N. Huang, M. E. Hurles, H. Jin, L. Jostins, T. M. Keane, S. Q. Le, S. Lindsay, Q. Long, D. G. MacArthur, S. B. Montgomery, L. Parts, J. Stalker, C. Tyler-Smith, K. Walter, Y. Zhang, M. B. Gerstein, M. Snyder, A. Abyzov, S. Balasubramanian, R. Bjornson, J. Du, F. Grubert, L. Habegger, R. Haraksingh, J. Jee, E. Khurana, H. Y. Lam, J. Leng, X. J. Mu, A. E. Urban, Z. Zhang, Y. Li, R. Luo, G. T. Marth, E. P. Garrison, D. Kural, A. R. Quinlan, C. Stewart, M. P. Stromberg, A. N. Ward, J. Wu, C. Lee, R. E. Mills, X. Shi, S. A. McCarroll, E. Banks, M. A. DePristo, R. E. Handsaker, C. Hartl, J. M. Korn, H. Li, J. C. Nemesh, J. Sebat, V. Makarov, K. Ye, S. C. Yoon, J. Degen. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, Oct 2010.

[3] W. J. Ansorge. Next-generation DNA sequencing techniques. *N Biotechnol*, 25(4):195–203, Apr 2009.

[4] No authors listed. The ENCODE (ENCyclopedia Of DNA Elements) Project. *Science*, 306(5696):636–640, Oct 2004.

[5] Boehm B., Rombach H.D., and Zelkowitz M.V., editors. *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer, 2005.

[6] W. Bialek and D. Botstein. Introductory science and mathematics education for 21st-Century biologists. *Science*, 303(5659):788–790, Feb 2004.

[7] F. S. Collins, M. Morgan, and A. Patrinos. The Human Genome Project: lessons from large-scale biology. *Science*, 300(5617):286–290, Apr 2003.

[8] Heaton D., Carver J. C., Barlett R., Oakes K., and Hochstein L. The relationship between development problems and use of software engineering practices in computational science and engineering: a survey. website, 2012. `http://www.software.ac.uk/sites/default/files/softwarepractice2012_submission_10.pdf`.

[9] Kelly D. and Sanders R. Assessing the Quality of Scientific Software. website, 2008. `http://secse08.cs.ua.edu/Papers/Kelly.pdf`.

[10] S. A. Forbes, N. Bindal, S. Bamford, C. Cole, C. Y. Kok, D. Beare, M. Jia, R. Shepherd, K. Leung, A. Menzies, J. W. Teague, P. J. Campbell, M. R. Stratton, and P. A. Futreal. COSMIC: mining complete cancer genomes in the Catalogue of Somatic Mutations in Cancer. *Nucleic Acids Res.*, 39(Database issue):D945–950, Jan 2011.

[11] Wilson G. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1), 2006.

[12] D. C. Ince, L. Hatton, and J. Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, Feb 2012.

[13] Segal J. Sone problems of professional end user developers. *In: IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007.

[14] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, J. Howland, L. Kann, J. Lehoczky, R. LeVine, P. McEwan, K. McKernan, J. Meldrim, J. P. Mesirov, C. Miranda, W. Morris, J. Naylor, C. Raymond, M. Rosetti, R. Santos, A. Sheridan, C. Sougnez, N. Stange-Thomann, N. Stojanovic, A. Subramanian, D. Wyman, J. Rogers, J. Sulston, R. Ainscough, S. Beck, D. Bentley, J. Burton, C. Clee, N. Carter, A. Coulson, R. Deadman, P. Deloukas, A. Dunham, I. Dunham, R. Durbin, L. French, D. Grafham, S. Gregory, T. Hubbard, S. Humphray, A. Hunt, M. Jones, C. Lloyd, A. McMurray, L. Matthews, S. Mercer, S. Milne, J. C. Mullikin, A. Mungall, R. Plumb, M. Ross, R. Shownkeen, S. Sims, R. H. Waterston, R. K. Wilson, L. W. Hillier, J. D. McPherson, M. A. Marra, E. R. Mardis, L. A. Fulton, A. T. Chinwalla, K. H. Pepin, W. R. Gish, S. L. Chissoe, M. C. Wendl, K. D. Delehaunty, T. L. Miner, A. Delehaunty, J. B. Kramer, L. L. Cook, R. S. Fulton, D. L. Johnson, P. J. Minx, S. W. Clifton, T. Hawkins, E. Branscomb, P. Predki, P. Richardson, S. Wenning, T. Slezak, N. Doggett, J. F. Cheng, A. Olsen, S. Lucas, C. Elkin, E. Uberbacher, M. Frazier, R. A. Gibbs, D. M. Muzny, S. E. Scherer, J. B. Bouck, E. J. Sodergren, K. C. Worley, C. M. Rives, J. H. Gorrell, M. L. Metzker, S. L. Naylor, R. S. Kucherlapati, D. L. Nelson, G. M. Weinstock, Y. Sakaki, A. Fujiyama, M. Hattori, T. Yada, A. Toyoda, T. Itoh, C. Kawagoe, H. Watanabe, Y. Totoki, T. Taylor, J. Weissenbach, R. Heilig, W. Saurin, F. Artiguenave, P. Brottier, T. Bruls, E. Pelletier, C. Robert, P. Wincker, D. R. Smith, L. Doucette-Stamm, M. Rubenfield, K. Weinstock, H. M. Lee, J. Dubois, A. Rosenthal, M. Platzer, G. Nyakatura, S. Taudien, A. Rump, H. Yang, J. Yu, J. Wang, G. Huang, J. Gu, L. Hood, L. Rowen, A. Madan, S. Qin, R. W. Davis, N. A. Federspiel, A. P. Abola, M. J. Proctor, R. M. Myers, J. Schmutz, M. Dickson, J. Grimwood, D. R. Cox, M. V. Olson, R. Kaul, C. Raymond, N. Shimizu, K. Kawasaki, S. Minoshima, G. A. Evans, M. Athanasiou, R. Schultz, B. A. Roe, F. Chen, H. Pan, J. Ramser, H. Lehrach, R. Reinhardt, W. R. McCombie, M. de la Bastide, N. Dedhia, H. Blocker, K. Hornischer, G. Nordsiek, R. Agarwala, L. Aravind, J. A. Bailey, A. Bateman, S. Batzoglou, E. Birney, P. Bork, D. G. Brown, C. B. Burge, L. Cerutti, H. C. Chen, D. Church, M. Clamp, R. R. Copley, T. Doerks, S. R. Eddy, E. E. Eichler, T. S. Furey, J. Galagan, J. G. Gilbert, C. Harmon, Y. Hayashizaki, D. Haussler, H. Hermjakob, K. Hokamp, W. Jang, L. S. Johnson, T. A. Jones, S. Kasif, A. Kaspryzk, S. Kennedy, W. J. Kent, P. Kitts, E. V. Koonin, I. Korf, D. Kulp, D. Lancet, T. M. Lowe, A. McLysaght, T. Mikkelsen, J. V. Moran, N. Mulder, V. J. Pollara, C. P. Ponting, G. Schuler, J. Schultz, G. Slater, A. F. Smit, E. Stupka, J. Szustakowski, D. Thierry-Mieg, J. Thierry-Mieg, L. Wagner, J. Wallis, R. Wheeler, A. Williams, Y. I. Wolf, K. H. Wolfe, S. P. Yang, R. F. Yeh, F. Collins, M. S. Guyer, J. Peterson, A. Felsenfeld, K. A. Wetterstrand, A. Patrinos, M. J. Morgan, P. de Jong, J. J. Catanese, K. Osoegawa, H. Shizuya, S. Choi, Y. J. Chen, and J. Szustakowki. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, Feb 2001.

[15] C. M. Lewis, A. Obregon-Tito, R. Y. Tito, M. W. Foster, and P. G. Spicer. The Human Microbiome Project: lessons from human genomics. *Trends Microbiol.*, 20(1):1–4, Jan 2012.

[16] E. R. Mardis. Anticipating the 1,000 dollar genome. *Genome Biol.*, 7(7):112, 2006.

[17] G. Miller. Scientific publishing. A scientist's nightmare: software problem leads to five retractions. *Science*, 314(5807):1856–1857, Dec 2006.

[18] A. Morin, J. Urban, P. D. Adams, I. Foster, A. Sali, D. Baker, and P. Sliz. Research priorities. Shining light into black boxes. *Science*, 336(6078):159–160, Apr 2012.

[19] R. M. Myers, J. Stamatoyannopoulos, M. Snyder, I. Dunham, R. C. Hardison, B. E. Bernstein, T. R. Gingeras, W. J. Kent, E. Birney, B. Wold, G. E. Crawford, B. E. Bernstein, C. B. Epstein, N. Shoresh, J. Ernst, T. S. Mikkelsen, P. Kheradpour, X. Zhang, L. Wang, R. Issner, M. J. Coyne, T. Durham, M. Ku, T. Truong, L. D. Ward, R. C. Altshuler, M. F. Lin, M. Kellis, T. R. Gingeras, C. A. Davis, P. Kapranov, A. Dobin, C. Zaleski, F. Schlesinger, P. Batut, S. Chakrabortty, S. Jha, W. Lin, J. Drenkow, H. Wang, K. Bell, H. Gao, I. Bell, E. Dumais, J. Dumais, S. E. Antonarakis, C. Ucla, C. Borel, R. Guigo, S. Djebali, J. Lagarde, C. Kingswood, P. Ribeca, M. Sammeth, T. Alioto, A. Merkel, H. Tilgner, P. Carninci, Y. Hayashizaki, T. Lassmann, H. Takahashi, R. F. Abdelhamid, G. Hannon, K. Fejes-Toth, J. Preall, A. Gordon, V. Sotirova, A. Reymond, C. Howald, E. Graison, J. Chrast, Y. Ruan, X. Ruan, A. Shahab, W. Ting Poh, C. L. Wei, G. E. Crawford, T. S. Furey, A. P. Boyle, N. C. Sheffield, L. Song, Y. Shibata, T. Vales, D. Winter, Z. Zhang, D. London, T. Wang, E. Birney, D. Keefe, V. R. Iyer, B. K. Lee, R. M. McDaniell, Z. Liu, A. Battenhouse, A. A. Bhinge, J. D. Lieb, L. L. Grasfeder, K. A. Showers, P. G. Giresi, S. K. Kim, C. Shestak, R. M. Myers, F. Pauli, T. E. Reddy, J. Gertz, E. C. Partridge, P. Jain, R. O. Sprouse, A. Bansal, B. Pusey, M. A. Muratet, K. E. Varley, K. M. Bowling, K. M. Newberry, A. S. Nesmith, J. A. Dilocker, S. L. Parker, L. L. Waite, K. Thibeault, K. Roberts, D. M. Absher, B. Wold, A. Mortazavi, B. Williams, G. Marinov, D. Trout, S. Pepke, B. King, K. McCue, A. Kirilusha, G. DeSalvo, K. Fisher-Aylor, H. Amrhein, J. Vielmetter, G. Sherlock, A. Sidow, S. Batzoglou, R. Rauch, A. Kundaje, M. Libbrecht, E. H. Margulies, S. C. Parker, L. Elnitski, E. D. Green, T. Hubbard, J. Harrow, S. Searle, F. Kokocinski, B. Aken, A. Frankish, T. Hunt, G. Despacio-Reyes, M. Kay, G. Mukherjee, A. Bignell, G. Saunders, V. Boychenko, M. Van Baren, R. H. Brown, E. Khurana, S. Balasubramanian, Z. Zhang, H. Lam, P. Cayting, R. Robilotto, Z. Lu, R. Guigo, T. Derrien, A. Tanzer, D. G. Knowles, M. Mariotti, W. James Kent, D. Haussler, R. Harte, M. Diekhans, M. Kellis, M. Lin, P. Kheradpour, J. Ernst, A. Reymond, C. Howald, E. A. Graison, J. Chrast, M. Tress, J. M. Rodriguez, M. Snyder, S. G. Landt, D. Raha, M. Shi, G. Euskirchen, F. Grubert, M. Kasowski, J. Lian, P. Cayting, P. Lacroute, Y. Xu, H. Monahan, D. Patacsil, T. Slifer, X. Yang, A. Charos, B. Reed, L. Wu, R. K. Auerbach, L. Habegger, M. Hariharan, J. Rozowsky, A. Abyzov, S. M. Weissman, M. Gerstein, K. Struhl, N. Lamarre-Vincent, M. Lindahl-Allen, B. Miotto, Z. Moqtaderi, J. D. Fleming, P. Newburger, P. J. Farnham, S. Frietze, H. O'Geen, X. Xu, K. R. Blahnik, A. R. Cao, S. Iyengar, J. A. Stamatoyannopoulos, R. Kaul, R. E. Thurman, H. Wang, P. A. Navas, R. Sandstrom, P. J. Sabo, M. Weaver, T. Canfield, K. Lee, S. Neph, V. Roach, A. Reynolds, A. Johnson, E. Rynes, E. Giste, S. Vong, J. Neri, T. Frum, E. M. Johnson, E. D. Nguyen, A. K. Ebersol, M. E. Sanchez, H. H. Sheffer, D. Lotakis, E. Haugen, R. Humbert, T. Kutyavin, T. Shafer, J. Dekker, B. R. Lajoie, A. Sanyal, W. James Kent, K. R. Rosenbloom, T. R. Dreszer, B. J. Raney, G. P. Barber, L. R. Meyer, C. A. Sloan, V. S. Malladi, M. S. Cline, K. Learned, V. K. Swing, A. S. Zweig, B. Rhead, P. A. Fujita, K. Roskin, D. Karolchik, R. M. Kuhn, D. Haussler, E. Birney, I. Dunham, S. P. Wilder, D. Keefe, D. Sobral, J. Herrero, K. Beal, M. Lukk, A. Brazma, J. M. Vaquerizas, N. M. Luscombe, P. J. Bickel, N. Boley, J. B. Brown, Q. Li, H. Huang, M. Gerstein, L. Habegger, A. Sboner, J. Rozowsky, R. K. Auerbach, K. Y. Yip, C. Cheng, K. K. Yan, N. Bhardwaj, J. Wang, L. Lochovsky, J. Jee, T. Gibson, J. Leng, J. Du, R. C. Hardison, R. S. Harris, G. Song, W. Miller,

D. Haussler, K. Roskin, B. Suh, T. Wang, B. Paten, W. S. Noble, M. M. Hoffman, O. J. Buske, Z. Weng, X. Dong, J. Wang, H. Xi, S. A. Tenenbaum, F. Doyle, L. O. Penalva, S. Chittur, T. D. Tullius, S. C. Parker, K. P. White, S. Karmakar, A. Victorsen, N. Jameel, N. Bild, R. L. Grossman, M. Snyder, S. G. Landt, X. Yang, D. Patacsil, T. Slifer, J. Dekker, B. R. Lajoie, A. Sanyal, Z. Weng, T. W. Whitfield, J. Wang, P. J. Collins, N. D. Trinklein, E. C. Partridge, R. M. Myers, M. C. Giddings, X. Chen, J. Khatun, C. Maier, Y. Yu, H. Gunawardena, B. Risk, E. A. Feingold, R. F. Lowdon, L. A. Dillon, P. J. Good, J. Harrow, and S. Searle. A user's guide to the encyclopedia of DNA elements (ENCODE). *PLoS Biol.*, 9(4):e1001046, Apr 2011.

[20] NIAID and NHGRI. Eukaryotic Pathogen and Disease Vector Sequencing Project. Website, 2012. `http://www.niaid.nih.gov/labsandresources/resources/dmid/gsc/pathogen/Pages/default.aspx`.

[21] NIEHS. Environmental Genome Project. Website, 2012. `http://egp.gs.washington.edu/`.

[22] NIH. Centers for Mendelian Genomics. Website, 2012. `http://www.genome.gov/27546192`.

[23] NIH. Clinical Sequencing Exploratory Research. Website, 2012. `http://www.genome.gov/27546194`.

[24] NIH. International HapMap Project. Website, 2012. `http://hapmap.ncbi.nlm.nih.gov/`.

[25] no authors listed. Free Software Foundation. Website, 2012. `http://www.fsf.org/`.

[26] U.S. Department of Energy. Human Genome Project Information. Website, 2012. `http://www.ornl.gov/sci/techresources/Human\_Genome/home.shtml`.

[27] P. Pevzner and R. Shamir. Computing has changed biology–biology education must catch up. *Science*, 325(5940):541–542, Jul 2009.

[28] B. Ragan-Kelley, W. A. Walters, D. McDonald, J. Riley, B. E. Granger, A. Gonzalez, R. Knight, F. Perez, and J. G. Caporaso. Collaborative cloud-enabled tools allow rapid, reproducible biological insights. *ISME J*, Oct 2012.

[29] B. J. Strasser and M. O. Dayhoff. Collecting, comparing, and computing sequences: the making of Margaret O. Dayhoff's Atlas of Protein Sequence and Structure, 1954-1965. *J Hist Biol*, 43(4):623–660, 2010.

[30] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Katy Huff, Ian Mitchell, Mark Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *arXiv*, abs/1210.0530, 2012.

[31] Merali Z. ...why scientific programming does not compute. *Nature*, 467:775–7, Oct 2010.

[32] I. B. Zhulin. It is computation time for bacteriology! *J. Bacteriol.*, 191(1):20–22, Jan 2009.