Summary of Contents

Contents

CHAPTER 0

Introduction: Building a Computing Toolbox

No matter how much time you spend in the field or at the bench, most of your research is done when sitting in front of a computer. Yet, the typical curriculum of a biology PhD does not include much training on how to use these machines. It is assumed that students will figure things out by themselves, unless they join a laboratory devoted to computational biology—in which case they will likely be trained by other members of the group in the laboratory's (often idiosyncratic) selection of software tools. But for the vast majority of academic biologists, these skills are learned the hard way—through painful trial and error, or during long sessions sitting with the one student in the program who is "good with computers."

This state of affairs is at odds with the enormous growth in the size and complexity of data sets, as well as the level of sophistication of the statistical and mathematical analysis that goes into a modern scientific publication in biology. If, once upon a time, coming up with an original idea and collecting great data meant having most of the project ready, today the data and ideas are but the beginning of a long process, culminating in publication.

The goal of this book is to build a basic computational toolbox for biologists, useful both for those doing laboratory and field work, and for those with a computational focus. We explore a variety of tools and show how they can be integrated to construct complex pipelines for automating data collection, storage, analysis, visualization, and the preparation of manuscripts ready for submission.

These tools are quite disparate and can be thought of as LEGO® bricks, that can be combined in new and creative ways. Once you have added a new tool to your toolbox, the potential for new research is greatly expanded. Not only will you be able to complete your tasks in a more organized, efficient, and reproducible way, but you will attempt answering new questions that would have been impossible to tackle otherwise.

0.1 The Philosophy

Fundamentally, this book is a manifesto for a certain approach to computing in biology. Here are the main points we want to emphasize:

Automation

Doing science involves repeating the same tasks several times. For example, you might need to repeat an analysis when new data are added, or if the same analysis needs to be carried out on separate data sets, or again if the reviewers ask you to change this or that part of the analysis to make sure that the results are robust.

In all of these cases you would like to automate the processing of the data, such that the data organization and analysis and the production of figures and statistical results can be repeated without any effort. Throughout the book, we keep automation at the center of our approach.

Reproducibility

Science should be reproducible, and much discussion and attention goes into carefully documenting empirical experiments so that they can be repeated. In theory, reproducing statistical analysis or simulations should be much easier, provided that the data and parameters are available. Yet, this is rarely the case—especially when the processing of the data involves clicking one's way through a graphical interface without documenting all the steps. In order to make it easy to reproduce your results, your computational work should be

readable: Your analysis should be easy to read and understand. This involves writing good code and documenting what you are doing. The best way to proceed is to think of your favorite reader: yourself, six months from now. When you receive feedback from the reviewers, and you have to modify the analysis, will you be able to understand precisely what you did, how, and why? Note that there is no way to email yourself in the past to ask for clarifications.

organized: Keeping the project tidy and well organized is a struggle, but you don't want to open your project directory only to find that there are 16 versions of the same program, all with slight—and undocumented—variations!

self-contained: Ideally, you want all of your data, code, and results in the same place, without dependencies on other files or code that are not in the same location. In this way, it is easy to share your work with others, or to work on your projects from different computers.

Openness

Science is a worldwide endeavor. If you use costly, proprietary software, the chances are that researchers in less fortunate situations cannot reproduce your results or use your methods to analyze their data. Throughout the book, we focus on *free software*:[1] not only is the software free in the sense that it costs nothing, but free also means that you have the *freedom* to run, copy, distribute, study, change, and improve the software.

Simplicity

Try to keep your analysis as simple as possible. Sometimes, "readable" and "clever" are at odds, meaning that a single line of code processing data in 14 different ways at once might be genius, but seldom is it going to be readable. In such cases, we tend to side with readability and simplicity—even if this means writing three additional lines of code. We also advocate the use of plain text whenever possible, as text is portable to all computer architectures and will be readable decades from now.

Correctness

Your analysis should be correct. This means that programming in science is very different from programming in other areas. For example, *bugs* (errors in the code) are something the software industry has learned to manage and live with—if your application unexpectedly closes or if your word processor sometimes goes awry, it is surely annoying, but unless you are selling pacemakers this is not going to be a threat. In science, it is essential that your code does solely what it is meant to do: otherwise your results might be unjustified. This strong emphasis on correctness is peculiar to science, and therefore you

1. gnu.org/philosophy/free-sw.html.

will not find all of the material we present in a typical programming textbook. We explore basic techniques meant to ensure that your code is correct and we encourage you to rewrite the same analysis in (very) different programming languages, forcing you to solve the problem in different ways; if all programs yield exactly the same results, then they are probably correct.

Science as Software Development

There is a striking parallel between the process of developing software and that of producing science. In fact, we believe that basic tools adopted by software developers (such as version control) can naturally be adapted to the world of research. We want to build software pipelines that turn ideas and data into published work; the development of such a pipeline has important milestones, which parallel those of software development: one can think of a manuscript as a "beta version" of a paper, and even treat the comments of the reviewers as bugs in the project which we need to fix before releasing our product! The development of these pipelines is another central piece of our approach.

0.2 The Structure of the Book

The book is composed of 10 semi-independent chapters:

Chapter 1: Unix
: We introduce the Unix command line and show how it can be used to automate repetitive tasks and "massage" your data prior to analysis.

Chapter 2: Version control
: Version control is a way to keep your scientific projects tidily organized, collaborate on science, and have the whole history of each project at your fingertips. We introduce this topic using Git.

Chapter 3: Basic programming
: We start programming, using Python as an example. We cover the basics: from assignments and data structures to the reading and writing of files.

Chapter 4: Writing good code
: When we write code for science, it has to be correct. We show how to organize your code in an effective way, and introduce debugging, unit testing, and profiling, again using Python.

Chapter 5: Regular expressions

When working with text, we often need to find snippets of text matching a certain "pattern." Regular expressions allow you to describe to a computer what you are looking for. We show how to use the Python module `re` to extract information from text.

Chapter 6: Scientific computing

Modern programming languages offer specific libraries and packages for performing statistics, simulations, and implementing mathematical models. We briefly cover these tools using Python. In addition, we introduce `Biopython`, which facilitates programming for molecular biology.

Chapter 7: Scientific typesetting

We introduce LaTeX for scientific typesetting of manuscripts, theses, and books.

Chapter 8: Statistical computing

We introduce the statistical software R, which is fully programmable and for which thousands of packages written by scientists for scientists are available.

Chapter 9: Data wrangling and visualization

We introduce the `tidyverse`, a set of R packages that allow you to write pipelines for the organization and analysis of large data sets. We also show how to produce beautiful figures using `ggplot2`.

Chapter 10: Relational Databases

We present relational databases and `sqlite3` for storing and working efficiently with large amounts of data.

Clearly, there is no way to teach these computational tools in 10 brief chapters. In fact, in your library you will find several thick books devoted to each and every one of the tools we are going to explore. Similarly, becoming a proficient programmer cannot be accomplished by reading a few pages, but rather it requires hundreds of hours of practice. So why try to cover so much material instead of concentrating on a few basic tools?

The idea is to provide a structured guide to help jump-start your learning process for each of these tools. This means that we emphasize breadth over depth (a very unusual thing to do in academia!) and that success strongly depends on your willingness to practice by trying your hand at the exercises and embedding these tools in your daily work. Our goal is to *showcase* each tool by first explaining what the tool is and why you should master it. This allows you to make an informed decision on whether to invest your time in learning how to use it. We then guide you through some basic

features and give you a step-by-step explanation of several simple examples. Once you have worked through these examples, the learning curve will appear less steep, allowing you to find your own path toward mastering the material.

0.2.1 How to Read the Book

We have written the book such that it can be read in the traditional way: start from the first page and work your way toward the end. However, we have striven to provide a modular structure, so that you can decide to skip some chapters, focus on only a few, or use the book as a quick reference.

In particular, the chapters on Unix (ch. 1), version control (ch. 2), LaTeX (ch. 7), and databases (ch. 10) can be read quite independently: you will sometimes find references to other chapters, but in practice there are no prerequisites. Also, you can decide to skip any of these chapters (though we love each of these tools!) without affecting the reading of the rest of the book.

We present programming in Python (chs. 3–6) and then again in R (chs. 8–9). While we go into more detail when explaining basic concepts in Python, you should be able to understand all of the R material without having read any of the other chapters. Similarly, if you do not plan to use R, you can skip these chapters without impacting the rest of the book.

0.2.2 Exercises and Further Reading

In each chapter, upon completion of the material you will be ready to start working on the "Exercises" section. One of the main features of this book is that exercises are based on real biological data taken from published papers. As such, these are not silly little exercises, but rather examples of the challenges you will overcome when doing research. We have seen that some students find this level of difficulty frustrating. It is entirely normal, however, to have no idea how to solve a problem at first. Whenever you feel that frustration is blocking your creativity and efficiency, take a short break. When you return, try breaking the problem into smaller steps, or start from a blank slate and attempt an entirely different approach. If you keep chipping away at the exercise, then little by little you will make sense of what the problem entails and—finally—you will find a way to crack it. Learning how to enjoy problem solving and to take pride in a job well done are some of the main characteristics of a good scientist.

For example, we are fond of this quote from Andrew Wiles (who proved the famous Fermat's last theorem, which baffled mathematicians for centuries): "You enter the first room of the mansion and it's completely dark. You stumble around bumping into the furniture but gradually you learn where each piece of furniture is. Finally, after six months or so, you find the light switch, you turn it on, and suddenly it's all illuminated."[2] Hopefully, it will take you less than six months to crack the exercises!

Note that there isn't "a" way to solve a problem, but rather a multitude of (roughly) equivalent ways to do so. Each and every approach is perfect, provided that the results are correct and that the solution is found in a reasonable amount of time. Thus, we encourage you to consult our solutions to the exercises only once you have solved them: Did we come up with the same idea? What are the advantages and disadvantages of these approaches? Even if you did not solve the task entirely, you have likely learned a lot more while trying, compared to reading through the solutions upon hitting the first stumbling block. To provide a further stepping stone between having no idea where to start and a complete solution, we provide a pseudocode solution of each exercise online: the individual steps of the solution are described in English, but no code is provided. This will give you an idea how to approach the problem, but you will need to come up with the code. From there, it is only a short way to tackling your very own research questions. You can find the complete solutions and the pseudocode at computingskillsforbiologists.com/exercises.

When solving the exercises, the internet is your friend. Finding help online is by no means considered "cheating." On the contrary, if you find yourself exploring additional resources, you are doing exactly the right thing! As with research, anything goes, as long as you can solve your problem (and give credit where credit is due). Consulting the many comprehensive online forums gives you a sense of how widespread these computational tools are. Keep in mind that the people finding clever answers to your questions also started from a blank slate at some point in their career. Moreover, seeing that somebody else asked exactly your question should further convince you that you are on the right track.

Last but not least, the "Reading" section of each chapter contains references to books, tutorials, and online resources to further the knowledge of the material. If the chapter is an appetizer, meant to whet your appetite for knowledge, the actual meal is contained in the reading list. This book is a road map that equips you with sufficient knowledge to choose the appropriate tool for each task, and take the guesswork out of "Where should I start

2. computingskillsforbiologists.com/provingfermat.

my learning journey?" However, only by reading more on the topic and by introducing these tools into your daily research work will you be able to truly master these skills, and make the most of your computer.

We conclude with the sales pitch we use to present the class that inspired this book. If you are a graduate student and you read the material, you work your way through all the exercises, constantly striving to further your knowledge of these topics by introducing them into your daily work, then you will shave six months off your PhD—and not *any* six months, but rather those spent wrestling with the data, repeating tedious tasks, and trying to convince the computer to be reasonable and spit out your thesis. All things considered, this book aims to make you a happier, more productive, and more creative scientist. Happy computing!

0.3 Use in the Classroom

We have been teaching the material covered in this book, in the graduate class Introduction to Scientific Computing for Biologists, at the University of Chicago since 2012. The enrollment has been about 30 students per year. We found the material appropriate for junior graduate students as well as senior undergraduates with some research experience.

The University of Chicago runs on a quarter system, allowing for 10 lectures of three hours each. Typically, each chapter is covered by a single lecture, with "Version Control" (ch. 2) and "Scientific Typesetting" (ch. 7) each taking about an hour and a half, and "Writing Good Code" (ch. 4) and "Statistical Computing" (ch. 8) taking more than one lecture each.

In all cases, we taught students who had computers available in class, either by teaching in a computer lab, or by asking students to bring their personal laptops. Rather than using slides, the instructor lectured while typing all the code contained in the book during the class. This makes for a very interactive class, in which all students type all of the code too—making sure that they understand what they are doing. Clearly, this also means that the pace is slowed down every time a student has included a typo in their commands, or cannot access their programs. To ease this problem, having teaching assistants for the class helps immensely. Students can raise their hand, or stick a red post-it on their computer to signal a problem. The teaching assistant can immediately help the student and interrupt the class in case the problem is shared by multiple students—signaling the need for a more general explanation.

To allow the class to run smoothly, each student should prepare their computer in advance. We typically circulate each chapter a week in advance of class, encouraging the students to (a) install the software needed for the class

and (b) read the material beforehand. Teaching assistants also offer weekly office hours to help with the installation of software, or to discuss the material and the exercises in small groups.

The "intermezzos" that are interspersed in each chapter function very well as small in-class exercises, allowing the students to solidify their knowledge, as well as highlighting potential problems with their understanding of the material.

We encourage the students to work in groups on the exercises at the end of each chapter, and review the solutions at the beginning of the following class. While this can cause some difficulties in grading, we believe that working in groups is essential to overcome the challenge of the exercises, making the students more productive, and allowing less experienced students to learn from their peers. Publishing a blog where each group posts their solutions reinforces the esprit de corps, creating a healthy competition between the groups, and further instilling in the students a sense of pride for a job well done. We also encouraged students to constructively comment on the different approaches of other groups and discuss the challenges they've faced while solving the exercises.

Another characteristic of our class has been the emphasis on the practical value of the material. For example, we ask each student to produce a final project in which they take a boring, time-consuming task in their laboratory (e.g., analysis of batches of data produced by laboratory machines, calibration of methods, other repetitive computational tasks) and completely automate it. The student then shows their work to their labmates and scientific advisor, and writes a short description of the program, along with the documentation necessary to use it. The goal of the final project is simply to show the student that mastering this material can save them a lot of time—even when accounting for the strenuous process of writing their first programs.

We have also experimented with a "flipped classroom" setting, with mixed results. In this case, the students read the material at their own pace, and work through all the small exercises contained in the chapter. The lecture is then devoted to working on the exercises at the end of each chapter. The lecturer guides the discussion on the strategies that can be employed to solve the problem, sketching pseudocode on the board, and eventually producing a fully fledged code on the computer. We have observed that, while this approach is very rewarding for students with some prior experience in programming, it is much less engaging for novices, who feel lost and out of touch with the rest of the class. Probably, this would work much better if the class size were small (less than 10 students).

Finally, we have found that leading by example serves as powerful motivation to students. We have always shown that we use the tools covered here for our own research. A well-placed anecdote on Git saving the day, or showing

how all the tables in a paper were automatically generated with a few lines of R, can go a long way toward convincing the students that their work studying the material will pay off over a lifetime.

0.4 Formatting of the Book

You will find all `commands` and the names of `packages` typeset in a fixed-width font. User-provided `[INPUT]` is capitalized and set between square brackets. To execute the commands, you do not need to reproduce such formatting. Within explanatory text, *technical terms* are presented in italics.

Throughout the book, we provide many code examples, enclosed in gray boxes and typeset using fixed-width fonts. All code examples are also provided on the companion website computingskillsforbiologists.com—but we encourage you to type all the code in by yourself: while this might feel slow and inefficient, the learning effect is stronger compared to simply copying and pasting, and only inspecting the result. Within the code examples, language-specific `commands` are highlighted in bold.

Within the code boxes, we try to keep lines short. When we cannot avoid a line that is longer than the width of the page we use the symbol ↪ to indicate that what follows should be typed in the same line as the rest.

0.5 Setup

Before you can start computing, you need to set up the environment, and download the data and the code.

What You Need

A computer: All the software we present here is free and can be installed with a few commands in Linux Ubuntu or Apple's OS X; we strive to provide guidance for Windows users. There are no specific hardware requirements. All the tools require relatively little memory and space on your hard drive.

Software: Each chapter requires installing specific software. We have collected detailed instructions guiding you through the installation of each tool at computingskillsforbiologists.com/setup.

A text editor: While working through the chapters, you will write a lot of code. Much will be written in the integrated development environments (IDEs) Jupyter and RStudio. Sometimes, however, you will need to write code in a text editor. We

encourage you to keep working with your favorite editor, if you already have one. If not, please choose an editor that can support syntax highlighting for Python, R, and LaTeX. There are many options to choose from, depending on your architecture and needs.[3]

Initial Setup

You can find instructions for the initial setup on our website at computing skillsforbiologists.com/setup. We have bundled all the data, code, exercises, and solutions in a single download. We strongly recommend that you save this directory in your home directory (see section 1.3.2).

3. computingskillsforbiologists.com/texteditors.

General Index

Italic pages refer to figures and tables