

# Chapter 20

## Helpful tools

Many tools exist to help scientists work computationally. In addition to both general purpose and domain-specific programming languages, a wide assortment of programs exist to accomplish specific tasks. We call attention to a number of tools in this chapter, with a particular focus on good practices when using them, good practices computationally and good practices scientifically.

### 20.1 Computational notebooks

Computational notebooks play an increasingly important role in scientific computing and working with data, network or otherwise. All data scientists should be familiar with their use and understand good practices for ensuring they are used appropriately.

Examples of computational notebooks are Jupyter, R Markdown, and perhaps the pioneer of the format, Mathematica. Unlike a normal program or script, a computational notebook divides its contents into a sequence of blocks, often called cells. Cells can be executed individually and any outputs that would normally be printed to the user's screen are instead inserted into the notebook itself. Further, notebooks allow for different types of cells: code cells or text cells, with the latter allowing the user to record important information, thoughts, etc. These text cells are really what makes a notebook a, well, notebook; often, they support rich formatting, hyperlinks, and mathematical typesetting.

The power of notebooks comes from their ability to weave together code, the results or outputs of code, and non-code writing. Notebooks implement a kind of literate programming [247].<sup>1</sup> The power and flexibility comes at a cost, however.

Some inherent aspects of the design of notebooks can lead users away from good programming practice. For example, the interactivity of notebooks, that cells can be edited and rerun, especially out of the order in which they are written, makes notebooks ripe for bugs. Often, it's a good idea to rerun the entire notebook top-to-bottom to make sure all the code still works—and this can suddenly reveal subtle mistakes due to notebook cells being out of order. Worse, this problem is pervasive enough that often

---

<sup>1</sup> Literate programming as advocated for by Knuth [247] puts prose writing and computer programming on equal footing.

times scientists won't rerun the notebook, taking the results as a finished artifact as is, and this leads to reproducibility issues as the intervening edits that got the out-of-order notebook to its final state are not present in the notebook itself. This latter point, this loss of information, harms notebooks-as-archives, despite archiving being one of a notebook's central purposes.<sup>2</sup> Indeed, while many factors contribute to reproducibility issues with code (Ch. 19), a large-scale study of Jupyter notebooks on GitHub [374] recently found that only 24% of publicly available notebooks executed without errors, and only 4% produced the same results!

In general, we consider notebooks to be an important tool for data scientists, but one whose strengths and weaknesses should be well understood. Use notebooks appropriately.

## 20.2 Pipelines

It is useful to think of computational work as being composed of pipelines: First, data are downloaded. Then, loaded into your program. Next, a processing step is applied, perhaps a filtering criteria is used to select an appropriate subset of data. Afterwards, apply a statistical method. Finally, compute summary statistics or perhaps make some plots.

Together, the linear sequence of these steps forms a pipeline that you have built. Will you find yourself repeating those steps, for instance if the data are updated? Will you need to create a new, similar but not identical pipeline for a future calculation?

Such situations arise often enough that a variety of tools have been introduced to create, maintain, and run these computational pipelines. By far the most popular, particularly in computational biology and bioinformatics, is *Snakemake* [252]. Snake-make goes beyond basic pipelines with an entire workflow management system. Using readable Python code, a workflow can be described or documented in Snakemake, and that workflow can be carried over from your local machine to remote servers to large compute clusters.

Workflows and pipelines also work extremely well at gluing together different computing tools. Need to combine Python code with some cutting-edge Julia or legacy-but-still-potent Fortran code? No worries, interoperate these languages within your pipeline. In fact, a key idea of the UNIX operating system (Sec. 20.4) is to build most functionality out of pipelines of small, single-purpose utilities. UNIX has pipes! Now you can use specialist languages where they are most suited, and glue them together in a pipeline.

---

<sup>2</sup> Other, more technical issues arise when storing notebooks in a version control system (Sec. 20.5). Rerunning an identical notebook and getting identical results may still lead to changes to the contents of the notebook, changes invisible to us as users but still present in the data. These changes need to be stored in the version control system and they make it difficult for users tracking the notebooks to easily understand whether changes to a notebook are important or not.

## 20.3 Working with remote computers

Modern personal computers are exceptionally, impressively powerful, but scientists still find themselves using large workstations and supercomputers to analyze data and perform computations. Typically, these computers are used remotely: you sit at your personal device (say, your laptop) and connect to the computer over a local network or the Internet.<sup>3</sup> The connection allows you to transfer files back and forth between your local and remote computers as well as execute commands and run programs on the remote computer. Somewhat recently, “cloud computing” services have sprung up which act in a similar way, though commercial providers often provide easy-to-use website interfaces. Yet fundamentally, while these providers hide the details of the computers, you are still connecting with and using a remote computer or set of computers.

### Graphical vs. text-based interaction

One way to work with a remote computer is through a tool such as VNC (Virtual Network Computing) or other variants of what is called “Remote Desktopting.” Remote desktops provide a graphical means of working with the remote computer. In essence, a window is open on your local computer inside of which is the graphical interface of the remote computer—its windows, desktop, files, and such. You can click and drag inside this window, type commands, and act in many ways as if you are actually sitting in front of the remote computer.

The second way to work with the remote is through a text-only interface, called a command prompt or command line interface. Here you have the ability to issue commands to the remote computer by typing them in only. (Historically, this was the only way to work with a computer, local or remote.)

The graphical user interface (GUI) and command line interface (CLI) approaches have advantages and disadvantages:

- GUIs over a network connection can be annoying to use as any delay introduced by sending the graphics over the network makes the computer feel sluggish.
- GUIs require more computing resources on the part of the remote computer. Often, remote computers are set up without running any graphical interfaces in order to use as much memory and computation for their tasks.
- CLIs have a steeper learning curve compared to a GUI, particularly if the remote computer is using a GUI you are already familiar with. However, for scientific work, CLIs are incredibly powerful and we encourage you to invest the time to learn them.

Nowadays, any computer you sit down and work at will be running a GUI. But you can quickly access a CLI inside your GUI using a “terminal” or shell program.<sup>4</sup> Mac

<sup>3</sup> We forget, but very early computers worked exclusively in such a manner: an operator sat at a device and communicated with a central “mainframe” computer. The difference was the device was not a computer or laptop but usually a teletype: an electric typewriter connected to a telephone line. Early computer terminals with video screens were actually called “glass teletypes.”

<sup>4</sup> Sometimes these programs are called *terminal emulators* because they mimic (emulate) within your computer an old-fashioned text terminal connected to a remote mainframe computer.

computers come with an application *Terminal.app*, Linux OS computers have a variety of such programs,<sup>5</sup> and Windows computers come with a program called Command Prompt (`cmd.exe`).<sup>6</sup>



Learn to use a CLI.

If you're not familiar with using a text interface for a computer, consider learning it. Complementing graphical interfaces, command line interfaces are powerful and efficient ways to productively use a computer.

The most common tasks when working with remote computers are (1) transferring files to and from the remote computer and (2) issuing commands and running programs on the remote computer. Both tasks require establishing a connection—hopefully a secure one!—between your local computer and the remote computer and often that connection is made with the same underlying method for both types of tasks.

## Secure connections

To log into a remote computer works exactly like logging into a website. You create or receive a username and password and then you provide those to access your account. However, unlike using a website, when working with a remote computer you may find yourself logging in many times during a work session, and it can become tiresome to keep reentering your login credentials. The solution is to set up a *passwordless login* using a matching pair of *keys*: a *public* key which is copied to the remote computer (and can be safely viewed by anyone) and a *private* key (which is kept to your local computer and should never be copied or shared<sup>7,8</sup>). Together these keys allow you to log into the remote computer quickly, invaluable when you are logging in many times during a work session.

How can you log in securely without a password? Using public–private key pairs, the local and remote computers perform some calculations behind-the-scenes to convince the remote computer you are who you say you are. When you first ask the remote to log in, the remote will create a unique “challenge question” and then use your public key, already on the remote, to encrypt that challenge question. The remote will then send the encrypted question to the local (since this is sent over the network, assume a bad actor may be able to see it). The local computer can then use your private key to decrypt the question, read it, and answer it. The answer is then sent back to the remote computer. Unless the system is insecure, only someone with the private key can answer the challenge question, so when the remote receives the correct answer, it knows the

<sup>5</sup> If you're using Linux you almost certainly already know this program.

<sup>6</sup> Recent versions of Windows come with *Windows Terminal* which can run multiple CLIs. Windows users may also be familiar with PuTTY (<https://www.putty.org>), a Windows program that enables access to another computer's CLI.

<sup>7</sup> You can add a password to your private key for extra security if you wish, although doing so removes the convenience of passwordless login.

<sup>8</sup> It's quite easy to make new public–private key pairs and best practice if you need to log into one remote from multiple local computers (like a laptop and a desktop) is to generate a separate pair on each local machine and copy both public keys to the remote. SSH (discussed shortly) will find which public key is needed automatically.

local computer holds the right private key and that it is safe to assume you can access the account. Lastly, now that the remote is convinced you have access, some additional key pairs are created and exchanged in such a way that a shared secret is available on both computers and that secret can be used by both computers to encrypt and decrypt any data passing between the two. (This step also occurs when using a password for authentication.) At this point you have a secure connection or “tunnel” between the two computers.<sup>9</sup>

✓ Save time by setting up secure, passwordless login for any remote computers you regularly use.

## File transfer

A CLI usually gives you commands to move or copy files. For example, on a UNIX-style computer (Sec. 20.4):

```
cp huri_ppi01.edgelist ~/archive/
```

will copy (cp) the file `huri_ppi01.edgelist` into a folder in your home directory (`~/`) called `archive`. In other words, we can transfer files within our local computer. But such commands also extend almost automatically to transfer files *between* computers:

```
scp huri_ppi01.edgelist remote_supercomputer.org:~/archive/
```

This command is nearly identical, with two differences:

**scp** Instead of using the copy program, we use the *secure copy* program (scp). The *secure* refers to its ability to send data over the network using a secure connection as discussed above.

**remote\_supercomputer.org:** The destination of the copy now begins with the address of the remote computer followed by a colon (:). We can also specify a username: `user@computer:/path/to/folder`. The CLI naturally incorporates network addresses into file and folder names using this syntax, and thus we don't need to change much to send data between computers.

## Issuing commands

Complementing the ability to send files to a computer you can access is the ability to log into that computer from a CLI. Most commonly this is done with a program called *ssh* (*Secure SHell*), although some alternatives exist such as *mosh*.<sup>10</sup> To do so, in a CLI window on your local computer you run the appropriate *ssh* command, then the contents of that CLI will show the CLI running on the remote computer. Any commands you type will actually be seamlessly running on the remote computer. Issuing a logoff

<sup>9</sup> We have of course skipped over significant technical details, in particular how the keys are computed and how the challenge question is created and answered. This use of key pairs is a form of *asymmetrical encryption*.

<sup>10</sup> <https://mosh.org/>

command on the remote will then terminate this connection and your CLI will return to that of your local computer.

SSH powers the connection by forming a secure tunnel between the two computers, through which commands and their results are sent back-and-forth. SSH also enables other commands such as `scp` discussed above. Indeed, most commands that send data between computers securely rely on the same library.

SSH comes with many additional commands to make it easier to use. `ssh-keygen` allows you to create and store public–private key pairs using different algorithms. `ssh-agent` provides additional security by managing control over unencrypted keys on your local computer, allowing you to keep a password for your local key but not have to reenter the password for every remote work session. You can also use a config file to save the usernames, addresses, and other details for all the remote computers (hosts) you’re working with. The config file is especially helpful for creating aliases, short abbreviations for longer login details, saving you from typing in a long computer name or other information every time you want to connect to a given computer. Very handy!

## 20.4 UNIX—I know this system

Fundamentally, our computational work is performed on computers and those computers are powered by operating systems (OSes). While there appears to be a plethora of OSes to choose from—Macs and PCs and Linux devices—in reality they fall into two main groups: Windows and UNIX-style.<sup>11</sup>

What do we mean by “UNIX-style”? At the dawn of computing and into the mid-twentieth century, it was common for each machine to have a custom-made operating system. Often the first thing you needed to do with your new computer was write your OS for it! Gradually this changed. In the late 1960s, Dennis Richie and Ken Thompson, computer scientists at Bell Labs, designed an OS for a slightly out-of-date machine they were allowed to use. This machine was quite underpowered, even for the time, and so care was needed to create a minimal, efficient OS for it. Under those constraints, Richie and Thompson designed what would eventually be UNIX.<sup>12, 13</sup>

Why do we care? At the time of writing, every major computer operating system except one<sup>14</sup> is a direct descendent of UNIX. This includes Linux, macOS by Apple, all major smartphone OSes, both iPhone and Android, and, importantly for us, nearly every major supercomputer. The very limitations that Richie and Thompson wrestled with led to an efficient, flexible, and modular operating system architecture that would

<sup>11</sup> We are skipping over lots of small cases such as real-time OSes designed to power vehicles and other potentially dangerous and expensive equipment. Such dedicated systems are of little interest to us here.

<sup>12</sup> The name “UNIX” is a bit of a jab at a competing OS effort called “multics” coming from MIT. Multics was intended for multiple users on a large central mainframe computer. The original computer UNIX was designed for couldn’t support multiple users.

<sup>13</sup> In order to create UNIX, Richie and Thompson also needed to create a programming language to write it in. What they created would eventually be called C, and in many respects C is the foundation of most modern computing languages, either in design or implementation. Windows, the one major OS today not descended from UNIX, is written in C, C++, and C#, all languages that owe their existence to Richie and Thompson.

<sup>14</sup> That OS is Windows by Microsoft, a big exception as it is extremely popular for PCs. However, it is unheard of in other contexts, such as supercomputers, and, further, Microsoft has begun providing UNIX-style compatibility with the WSL (Windows Subsystem for Linux) initiative.

eventually form the basis for nearly all major OSes. But, besides very good design, there is another reason why UNIX propagated throughout computing. AT&T, then owner of Bell Labs, had an agreement with the United States government where it would not enter into any computing businesses in return for being allowed to maintain a monopoly on telephones and telecommunication. This meant AT&T could never sell UNIX and had to give it away.<sup>15</sup> People like free, especially cash-strapped computer scientists and small startup companies. Thus, over time, UNIX became the starting point for many operating systems, including “BSD,” which led to macOS, and Linux, which led to Android. We use the term “UNIX-style” for these UNIX descendants, which conform more or less to the major design patterns of UNIX.<sup>16</sup>

**i** Thanks to good design and a quirk of history, UNIX has become the foundation for nearly all computer operating systems, especially those used for scientific computing.

UNIX-style OSes have become the standard for scientific computing. If you ever plan to use a supercomputer (which nowadays is really nothing more than a large number of regular computers) or a cloud computing service, you will be using, at some level, a UNIX-style operating system. Take advantage of this by understanding and embracing some of its properties in your workflows.

**✓** Consider even adopting a UNIX-style OS for your personal work machine.

The modular design of UNIX-style OSes provides us several features to help with scientific workflows:

- Pipelines become natural. It is easy to make reusable programs that interoperate (Sec. 20.2).
- Remote access is easy. The OS’s modular nature includes separating interaction and display, making it easy to have the keyboard and screen come from a different, over-the-network computer. This gives us SSH (Sec. 20.3).
- Rethinking what it means to be an OS. Containers and virtual environments extend the next generation of scientific computing systems.

## Pipelines

UNIX-style OSes come with an elegant mechanism, the standard streams, for building pipelines that any program can take advantage of. The most important streams are `stdin` (read as: “standard in”) and `stdout` (read as: “standard out”).<sup>17</sup> In many ways, we can

<sup>15</sup> Given that not only UNIX but the *transistor*, the very basis of the information age, came out of Bell Labs, we are all very fortunate for this agreement.

<sup>16</sup> A codified standard operating system design called POSIX (ISO/IEC/IEEE 9945) is based on UNIX. For our purposes, “UNIX-style” is sufficient.

<sup>17</sup> We omit from this discussion `stderr`, the standard stream for reporting error messages. It is useful to separate error messages from output so that you can record them separately, in case you need to review what happened during a failed program run.

think of each as a file: your program can “read from” `stdin` in a manner exactly like reading from any other file, and likewise your program can “write to” `stdout`. What’s so nice about using these streams instead of normal files is that they can be composed. You can write one program that sends data to `stdout` and another program that receives data from `stdin`, then you can wire those programs together into a pipeline. Further, if your program is designed to use the standard streams, you can place it within any pipeline using other programs. All the programs that come with the OS already support these streams.

Here’s an example in Python using streams. If you are familiar with Python, the code for reading/writing streams is almost identical to reading/writing files:<sup>18</sup>

```
import sys

data_in_txt = sys.stdin.read()
result_txt = f(data_in_txt)
sys.stdout.write(result_txt)
```

This simple program mock-up receives data (as a string) from `stdin`, computes a result using a function `f(...)`, and then writes that result to `stdout`. (The streams work with text data because they are considered files; your program can convert `stdin`’s data as needed.) In Python, all you need to begin using standard streams is to import the system module (`import sys`).

Here’s a more specific example of a pipeline involving a text file and two programs. Suppose you’ve written a program, `get_nodes.py`, that uses `stdin` and `stdout`. We wish to send data to our program, pass our results to another program created by a colleague, then save the second program’s output to a file for storage. Here’s the command we would enter into our UNIX-style computer:

```
$ cat network.txt | python get_nodes.py19 |
  process_data > result.txt20
```

(The `$` is not typed in; we use it to represent the start of the “prompt” the computer displays showing us where to enter commands.) Besides the Python program we created, there are three things to understand in this pipeline: `cat`, `|`, and `>`.

**cat** `cat`, which stands for “concatenate,” is a standard UNIX program for printing a text file (`network.txt`). Printing it to what? In this case, to `stdout`, which becomes `stdin` for the next step of the pipeline, our Python program.

**|** The vertical bar or “pipe” character is, you guessed it, the symbol used to wire together the streams to build a pipeline. Here we used it to connect the text file to our `get_nodes.py` program, and then the output of our program to another command, `process_data`.

<sup>18</sup> As far as Python—and the OS—is concerned, the standard streams *are* files!

<sup>19</sup> Even better, you can set a flag inside your script called a “shebang” that will tell the computer to run it with Python automatically. Then the pipeline no longer specifically mentions Python. Why is this useful? If in the future you replace your Python script with something written in a different language, your pipeline code can remain unchanged!

<sup>20</sup> In practice, we recommend against generic names like `process_data` and `result.txt`.



- > A shorthand for redirecting stdout to a file. This lets us save `process_data` to a file of our choosing. While a program can always specify what output it will write, by taking output out of the program and putting into the pipeline, we can see and control this output ourselves without either modifying the program or, if the program lets us specify its output, learning how to do so.

This is just a taste of what UNIX-style pipes can do. In general, it's difficult to overstate the power of UNIX's simple, modular nature.

## 20.5 Version control

A version control system (VCS) allows multiple editors to work on shared files. Generally the users are programmers working together on a code base, but not always—this book, for instance, was written using just such a system.

The main problem a VCS addresses is handling conflicts: when two editors make different changes to the same part of a file, an algorithm cannot automatically merge those changes to make a single file. VCS systems provide functionality to pick out or replace the conflicting edits manually.

Allowing multiple editors to work separately on their own versions of a fileset also allows multiple versions of the fileset to be maintained. This is useful for software makers: you can maintain a “release” version of a package and a “beta” version of a future release at the same time, and use the VCS to switch between the versions as you work.

For researchers, version control is important for provenance: by keeping a detailed history of changes to the code, you can track who made what changes, when a particular output was first (or last) computed by the code, and you can revert to an older version of the code if you need to replicate exactly a past result. Scientists in particular benefit greatly from this use of a VCS.

**i** Git (<https://git-scm.com>) is by far the most popular VCS in use today. Git underlies GitHub (<https://github.com>) and many other open-source projects.

While not as user-friendly as others, it is the most important VCS to learn, and all programmers and data scientists should be familiar with the basics of using git.

A VCS provides a set of commands that examine and modify a *repository*. The repository, the central object of interest for the VCS, is just a folder on your computer, but besides your files, within it are special files that the VCS uses to record the full history of all the files placed under version control, or “tracked” by the VCS. (You need to tell the VCS specifically which files to track.) From these files, the VCS can efficiently build or rebuild the state of the repository at any point in its history; if you frequently tell the repository about your updates using relevant VCS commands (the repository is not updated automatically), then you will develop a fine-grained history for your project. Those special files also usually contain additional information, such as the addresses for copies of that repository on remote computers, if they have been set up. Saving these details makes it easy for you to update your local repository and then send your updates to the remote copy or, conversely, for you to quickly bring in any updates


from the remote repository to your local copy. Together, these commands, while they have a steep learning curve—it helps to know a bit *how* the VCS works to know *what* to do with it—smooth the road for productive work history and work collaboration.

## Version control for data?

Data scientists have a further need than programmers when it comes to version control: keeping the history of a dataset.

It may be that you are working with dynamic data, where new observations are constantly being added to a large dataset. Conversely, your data may come from a laborious process of data entry, where individuals manually code in observations. Data entry can be error-prone and may require processes whereby observations are verified and revised, if needed. In both cases, the data are changing, and a researcher should be able to review those changes as needed.

A VCS appears to be ideal for tracking dynamic data. After all, what's the difference between files that represent computer code and files that represent other data? As far as the computer is concerned, they're both ones and zeros, right? Unfortunately, there are differences, not in how the computer treats those files but in how the VCS was designed. Most VCSes use algorithms designed for small, human-readable, human-writeable files, such as source code. These algorithms don't handle very large files, or large changes to files, very well, and over time many common VCS tasks will become incredibly slow, even tasks not involving those large files.

 Version control is not well suited to large data files.

It is generally intended for files made by humans, such as source code. If you are generating large files (meaning, 100s of megabytes or more), particularly binary files, you probably want to avoid tracking their contents with a VCS. Doing so is likely to slow down the system considerably.

Some options do exist for using version control with data files. One is Git Large File Storage (LFS).<sup>21</sup> Git LFS is an extension you connect to git that allows git to track references to the large files and not the contents of the files themselves. Another option is Dolt,<sup>22</sup> a database that you can interact with as if it were a git repository. Hopefully, work in this area will continue and version control for data will become easier and more popular.

## 20.6 Backups

Backing up your data, writing, and other work against computer failure is critical. As discussed in Ch. 17, a good backup system accommodates both off-site backups and, more importantly, full file history. While most cloud computing services can provide off-site *copies*, they generally do not keep track of file history.

---

<sup>21</sup> <https://git-lfs.github.com>

<sup>22</sup> <https://github.com/dolthub/dolt>

For your personal computer, at the time of this writing, the two most popular file history backup services are Apple's Time Machine (for macOS computers) and Windows Backup (for Windows 10 and later). Both are free and built into their respective operating systems. Both also use a locally connected external disk<sup>23</sup> to store backups, although there is some support for backing up over a network. We strongly encourage users to use such a service if possible.

Version control systems such as git (Sec. 20.5) are an excellent choice for backing up the file histories of code and other "person-generated" files, so long as you set up a remote location to "push to" and you frequently add and update your work. While it would be great to use a similar system for tracking changing data, as we discussed, currently these systems are generally poor for large data files.

Research data backups are a different story. Your institution may provide facilities for data archiving and indeed there may be specific requirements for doing so. Be mindful of both legal requirements and ethical considerations.

## 20.7 Selecting tools for yourself

Two factors will continually drive your need to evaluate and re-evaluate new and existing computational tools. The first is obsolescence of software (and perhaps hardware). As tools fall out of favor, you may need to jump ship to alternatives. The second factor is the rise of new alternatives and innovations, plus, to an extent, changes in your needs as a researcher. Software is continually churning, with new ideas for solving problems and indeed new problems arising for us to solve.

A variety of tools exist to help you work computationally. Which do you wish to use? Do you have a problem that existing tools do not solve? Do you wish to improve your productivity by incorporating a new piece of software? We can help. Here we provide some advice on selecting a tool for a problem.

Suppose you have identified a problem you wish to solve. It may be a research question or a basic computing task. Ask yourself the following:

- Q0: Will an existing tool work on this problem? If so, how painful or painless is it as a solution?
- Q0.5: Is your problem even solved by a tool or not?
- Q1: What tools are available for this problem? If you are a beginner to the problem area, this question is easier said than addressed. Over time you will become more "plugged in" to the space of tools and options and this experience should be useful guidance.<sup>25</sup>

<sup>23</sup> A local disk does not provide an off-site backup. One option is to use two external disks<sup>24</sup> kept at different locations, for example, one in the office and one at home.

<sup>24</sup> Two is one and one is none.

<sup>25</sup> This underscores our assertion that good computing scientists need to always spend some portion of their time evaluating, re-evaluating, and changing their workflows. It seems highly efficient to identify a system that works, and never change it. But if you do need to make a change after being stuck for a long time, you will be out-of-practice at evaluating exactly these decisions and more likely to decide poorly. Take care to allot a portion of time, *but only a small portion*, to self-(re)evaluation.

Suppose now you have found a tool you wish to consider. Perhaps it is an open source project. Ask yourself:

- Q2: Does the tool solve your problem? Is it intended for exactly this problem or are you using it “off-label”?
- Q3: Is the tool popular?
- Q4: Is the tool actively maintained. Software goes stale fast!
- Q5: Is it new or well established with a long history?
- Q6: Is the tool open source or commercial? Are there restrictions on how you can use the tool?

Lastly, consider other criteria:

- Q7: Are you working with or accommodating others? Perhaps you work in a research group with specific requirements or workflows. Are you making decisions on a tool that others will have to use? Will they want to use the tool?
- Q8: Other criteria to evaluate? Cost? Compatibility?

## 20.8 Summary

We conclude our scientific computing interlude with a tour of some specific tools that we recommend at the time of this writing. These tools help scientists work with data in a reliable, reproducible, and hopefully efficient manner, saving time and perhaps even preventing mistakes.

Of course, the world of computing is always fast moving, and it is likely that specific tools mentioned here will fall out of favor and be replaced with new alternatives. A good, working computational scientist should always spend some (small) portion of their time evaluating current tools to see if better options exist; if you never take the time to switch up your workflow, and something comes along which forces a change such as a discontinued tool, you will be out of practice and may choose your new replacement poorly. To help, we have included some advice and a brief workflow to guide you through evaluating new tools to use.

### Bibliographic remarks

For readers interested in diving deeper into the world of practical computational tools and practices, we highly recommend *Effective Computation in Physics* by Scopatz and Huff [420]. Don’t let the title fool you, this isn’t just for physicists; it’s a tour of computing for any technically minded researcher interested in sharpening their scientific computing skills.

The most common form of cryptosystem for establishing secure connections between computers is based on the RSA algorithm, which creates challenge questions

by multiplying large prime numbers. For those wanting to learn more of the details, a brief overview of public key cryptography, also known as asymmetrical cryptography, by one of its founders, is given in an article by Hellman (of the Diffie–Hellman algorithm) [210].

A fantastic article demonstrating the power of UNIX pipes (written by their creator, Doug McIlroy) is Bentley et al. [52]. The program McIlroy discusses wonderfully illustrates the UNIX philosophy.

For those interested in learning more about UNIX’s creation, a recent memoir by Brian Kernighan, a major contributor to UNIX, provides an interesting account of its history [240]. More broadly, *The Idea Factory* by Gertner [180] gives an overarching story of Bell Labs, the research center where UNIX (and other major inventions) was developed.

Readers interested in learning to use a version control system should consult Scopatz and Huff [420] or the resources available at the git homepage.<sup>26</sup>

---

<sup>26</sup> <https://git-scm.com>

