Manuscript Title

This manuscript (permalink) was automatically generated from computer-aided-biotech/better-cb@c0d4c24 on February 15, 2021.

Authors

- **Benjamín J. Sánchez**
  0000-0001-6093-4110 · BenjaSanchez · BenjaSanchez
  Department of Bioengineering, Technical University of Denmark, Kgs. Lyngby, 2800, Denmark

- **Daniela C. Soto**
  0000-0002-6292-655X · dcsoto
  Genome Center, MIND Institute, and Department of Biochemistry & Molecular Medicine, Davis, CA 95616,USA

Abstract

This should be the abstract.

Introduction

Since Margaret Dayhoff pioneered the field of bioinformatics back in the sixties, the application of computational tools to the field of biology has vastly grown in scope and impact. Nowadays, biotechnological and biomedical research are routinely fed by the insights arising from novel computational approaches, machine learning algorithms and mathematical models. The ever increasing amount of biological data and the exponential growth in computing power will amplify this trend in the years to come.

The use of computers to address biological matters encompasses a wide array of applications usually grouped under the terms of “computational biology” and “bioinformatics”. Although distinct definitions have been delineated for each one [1,2], here we will consider both under the umbrella term “computational biology”, alluding to any application that involves the intersection of computing and biological data. As such, a computational biologist can be a data analyst, a data engineer, a statistician, a mathematical modeler, a software developer, and many others. In praxis, the modern computational biologist will be a “scientist of many hats”, taking on several of the duties listed above. But first and foremost, we will consider a computational biologist as a scientist whose ultimate goal is answering a biological question or addressing a need in the life sciences, by means of computation.

Scientific computing requires following specific practices to enable shareable, reproducible and sustainable outputs that stand the test of time. Computing-heavy disciplines such as software engineering and data science have already adopted practices addressing the need for collaboration, visualization, project management, and strengthening of online communities. However, as a highly interdisciplinary and evolving field, computational biology has yet to acquire a set of universal “best practices”. As most computational biologists come from diverse backgrounds and oftentimes lack formal computational training, the absence of guidelines can lead to disparate and unsustainable practices that hinder reproducibility and collaboration, and slow down biomedical and biotechnological research.

Over the last decade, several researchers have published advice directed to bench scientists starting in either scientific computing [3,4] or computational biology [5]. The advice encompasses a wide range of topics ranging from programming to project organization to manuscript writing. Other works have adopted a different approach, fixating in one powerful tool and diving deeper into its scope and applications, such as the software development and version control cloud service GitHub [6] and the web-application Jupyter Notebooks [7]. Similarly, recent works have chosen to comprehensively address one specific need in computational biology such as workflow automation [8] and software library development [9]. Although this advice proves immensely helpful, several aspects of the computational biology “journey” remain uncovered. Specifically, there is a lack of a clear roadmap regarding best practices from fundamental to advanced topics, in addition to practical examples tackling the complexities of this kind of research.

We premise that best practices in computational biology lie within a continuum that traverses three “levels”: the individual’s internal practices, the collaborative practices of a team, and the practices that allow a broader scientific community to access and engage over time with the research (Figure 1). Each one of these levels has a different set of needs and challenges. Here, we compile a selected list of relevant practices and related tools advisable for each one of these levels, emphasizing their time

and place in a computational biology research project. Finally, we illustrate the utility of these practices in three case studies covering the broader spectrum of computational biology research.

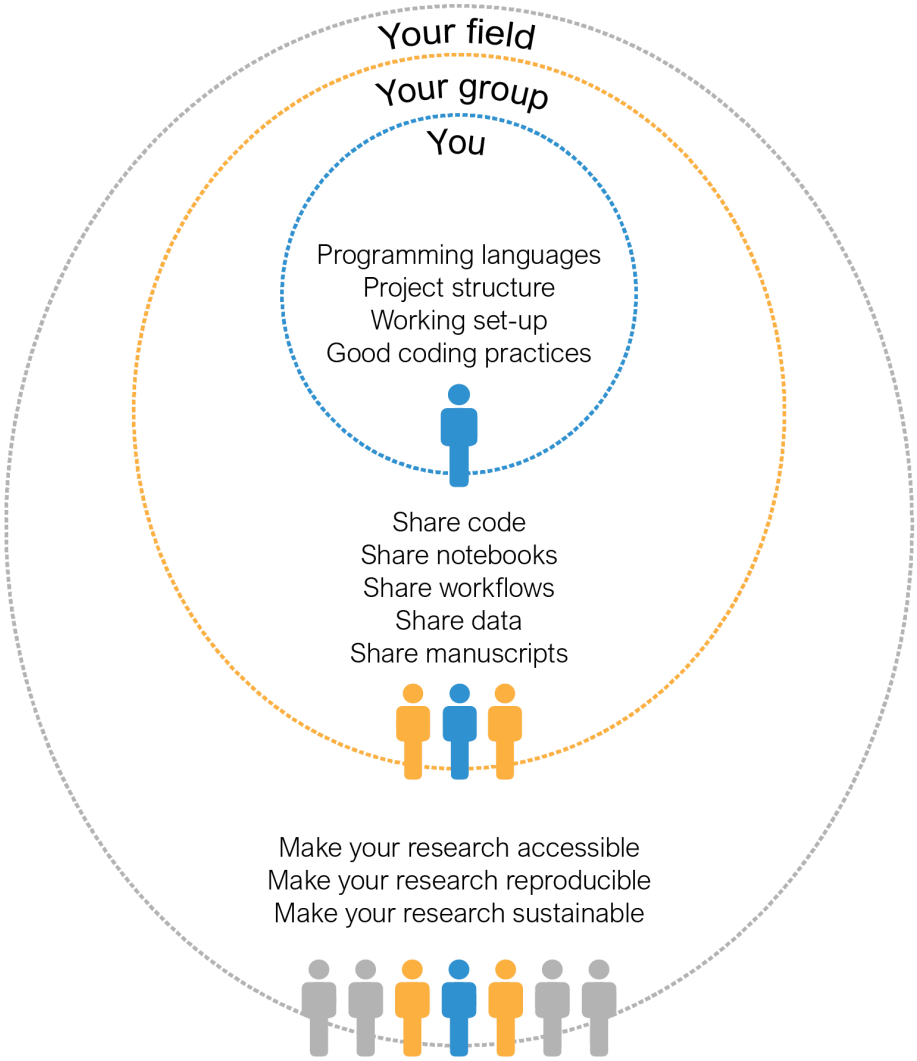

Figure 1: The three "levels" of computational biology include your personal research, your group and collaborators, and your scientific field.

Level 1: Personal Research

The computational biology "journey" begins with you—specifically, with the set of skills, tools and practices that you have put in place to conduct your research. Taking the time to optimally establish these building blocks will have high payoffs later, when you need to check your previous research (which will certainly happen). Consider that your most important collaborator is your future self, either of tomorrow or ten years from now. To consider all the aspects that make up a solid ground for any computational biology project, we devised a framework involving four main sequential steps (Table 1).

Step 1: Choose your programming languages

Different programming languages serve distinctive purposes and have different idiosyncrasies. As such, choosing the right programming language for a project depends on your research goals and personal preference or skill. Additionally, communities usually favor the usage and training of some

programming languages over others; utilizing such languages may facilitate integrating your work within the existing ecosystem.

As computational biology becomes a data intensive discipline, interacting with high-performance computing (HPC) operating systems has become a hallmark of the field. HPC infrastructures commonly use Unix/Linux distributions as their operating system. To interact with these platforms, you need to use a the command line interpreter known as the "Unix shell". There are multiple versions of Unix shells, being Bash one of the most widely adopted. Besides providing an "user interface", the shell is also a scripting language that allows manipulating files and executing programs through "shell scripts". Unix/Linux operating systems have other interesting perks, such as powerful and fast commands for searching words and manipulating files (e.g. `sed`, `grep` or `join`) as well as the language AWK, that can perform quick text processing, including arithmetic operations.

One of the most common task of any computational biologist is data analysis. The job of a data analyst involves data cleaning, exploration, manipulation, and visualization. Currently, Python is the most widely used programming language for data analysis worldwide [10,11]. Computational biology research has followed this trend, making Python one of the most popular languages among researchers. As machine learning and deep learning are more widely adopted in biological research, Python usage will keep growing. Python usage has been facilitated by the availability of packages for biological data analysis accessible through package managers such as Pip or Conda. Similarl to Python, R is the other prominent language in the field. Arguably, one of R main strengths is its wide array of tools for statistical analysis. Of particular interest is the Bioconductor repository, where many gold-standard tools for biological data analysis have been published. R usage in data science has deeply benefited from the Tidyverse packages, increasing the readability of the R syntax for both data manipulation via `dplyr`, and visualization via `ggplot2`.

Oftentimes, computational biologists require coding their own sets of instructions for processing data via scripts or programs. A script can be described as lightweight single-file program developed to tackle a narrow purpose. They are likely written in an interpreted programming language instead of a compiled one. Interpreted programming languages execute the program directly, without previous compilation, meaning each statement is run individually. They are quick to edit and can be run interactively, at expense of computational performance. In computational biology, the current most common multi-purpose scripting languages are Python and R. When working in a HPC, Shell/Bash scripting is also widely used. Perl and Matlab are also a popular language among bioinformatics and systems biology, respectively. A program, in the other hand, is a larger tool that usually combines multiple scripts and works as a "black box" to the user. It is designed to tackle computationally intensive problems, thus, a compiled language is preferred. Several tools designed for high-weight biological data processing have been written in C/C++. In recent years, however, scientists have been turning to Rust because of its speed, safety and friendly community [12]. If computational performance is not a concern, Python and R are suitable alternatives for coding programs.

Biological data processing is rarely a one-step process. To go from raw data to useful insights, several steps need to be taken in a specific order, accompanied by a plethora of decisions regarding parameters. Computational biologists have addressed this need by embracing workflow management systems to automate data analysis pipelines. A pipeline can be written as a Shell script where a handful of commands are written one after another, using Shell variables and Shell scripting syntax when needed. Although effective, this approach provides little control over the workflow, and lacks features to run isolated parts of the pipeline or track changes in input and output files. To overcome these limitations, a Bash script can be "upgraded" using the GNU Make program, which was originally designed to automate compilation and installation of software, but that it is flexible enough for building other types of workflows. Nowadays, however, several dedicated bioinformatics workflow managers have been developed. Snakemake is a workflow management system written in Python, allowing to incorporate the syntax of the tool with standard Python code. Similarly, Nextflow was build

as an extension of the Groovy—a programming language for the Java virtual machine—and can execute Groovy code. These tools not provide control over any step of the pipeline, but also offer features like interacting with job schedulers, software environment managers, and cloud computing support. Alternatively, workflows can be written in the Common Workflow Language (CWL)—a declarative standard to define workflows with the goal of enabling portability and reproducibility. Workflows written in CWL can be run in any CWL-enabled engine.

Step 2: Define your project structure

After choosing your programming languages and before starting any coding, we advice you to come up with a well-thought-out project structure. This design should be intentional and tailored to the present and future needs of your project—remember to be kind to your future self. A computational biology project requires, at the very least, a folder structure that supports code, data, and documentation. Although tempting, cramming all kind of files in a unique folder is unsustainable. Instead, separate each one on different folders and subfolders if needed. As additional principles consider documentation as a requirement and raw data as immutable. To simplify this process, you can base your project structure on research templates available off-the-rack. For data science projects, the python package Cookiecutter Data Science cut down the effort to the very minimum [13]. Running the package prompts a questionnaire in the terminal where you can input the project name, authors and other basic information. Then, the program generates a folder structure to store data—raw and processed—separated from notebooks and source code, as well as pre-made files for documentation such as a "readme", a docs folder and a license. Similarly, the Reproducible Research Project Initialization (rr-init) offers a template folder structure that can be cloned from a GitHub repository and modified by the user [14]. Although the later is simpler than the former, both follow an akin philosophy aimed at research correctness and reproducibility [15]. For workflow automation projects, we advice a similar folder structure. Snakemake recommends to store each workflow in a dedicated folder separated into workflow-related files—the Snakefile, rules and scripts—results and configuration [16]. In all cases, the folder must be initialize as a git repo for version control (See Step 4).

Beyond files and folders, the software and dependencies needed to run an analysis, workflow or program, are also part of the project structure itself. The intricacies of software installation and dependency management are not to be underestimated. Fortunately, package and virtual environment managers significantly reduce this burden. A package manager is a system that automates the installation, upgrading, configuration and removing of community-developed programs; a virtual environment manager, in the other hand, is a tool that generates isolated "environments" containing programs and dependencies that are functionally independent from other environments or the default operating system. Once a virtual environment is activated, a package manager can be used to install third-party programs.

We believe that a computational biology project must start with its own virtual environment. The main reason is reproducibility: environments save the project's dependencies and can restore them at will so the code can be run in any other computer. There are multiple options for both package and virtual environment management—some are language-specific; others, language-agnostic. If you are working with Python, you can initialize a Python environment using virtualenv or pipenv (where different Python versions can be installed). Inside the environment, you can use the Python package manager Pip to add Python code submitted to the Python Package Index (PyPI), GitHub or locally. For the R language, R-specific environments can be created using `renv`. After initializing the environment, packages can be installed via `install.packages` function from the Comprehensive R Archive Network (CRAN) and CRAN-like repositories. R also has BiocManager to install packages from the Bioconductor repository, which contains relevant software for high-throughput genomic sequencing analysis. To fully manage a dependencies beyond installation, RStudio developed the RStudio Package Manager which works with third-party code available in CRAN, Bioconductor, GitHub

or locally. A language-agnostic alternative is Conda—an increasingly popular package manager and a virtual environment manager. It supports program installation from the Anaconda repository, which contains the channel Bioconda specifically tailored to bioinformatics software. Also, if Python is installed, Python dependencies can be installed via pip. Conda is particularly helpful when working with third-part code in all sorts of languages—a common predicament for the computational biologist. Conda package and environment manager is included in both Miniconda and Anaconda distributions. The former is a minimal version of Anaconda, containing only Conda, Python, and a few useful packages.

Step 3: Choose your working set-up

With the foundation in place, the next step is to start coding. However, a more practical question needs to be answer first: where to code. The simplest tool available for this purpose are text editors. Since writing code is ultimately writing text, any tool where characters can be typed fulfills this purpose. However, coding can be streamlined by additional features as those found in code editors—text editors especially developed for writing code. Crucial features to facilitate coding include syntax highlight, indentation or autocompletion. Commonly used desktop editors include Atom, Sublime, Visual Studio Code, and Notepad++ (Windows only), all which a myriad of plugins available to enhance the coding experience. Command line text editors are also suitable options for coding, being Vim and Emacs the most powerful ones. All of these tools share the advantage of being language agnostic, allowing easy switching between languages, especially handy for the polyglot computational biologist.

In addition to text editors, integrated development environment (IDE) are also popular options for coding. As its essence, IDE are supercharged text editors, i.e. with multiple other features that make writing code easier. The main parts of an IDE are a code editor (with syntax highlight, indentation and suggestions), a debugger, a folder structure, and a way to execute your code (a compiler or interpreter). IDEs are not language agnostic, meaning that they allow to code in one language. The array of features also comes at a cost—IDEs usually use more memory and imply more visual clutter, if that is a concern of yours. For Python, Jupyter Lab, Spyder and JetBrains' PyCharm are popular options, while for R, RStudio is the gold-standard. Notably, the differences between an IDE and a code editor are somewhat blurry, especially when enough plugins have been added to a code editor.

In the latest years, notebooks have acquired relevance in computational biology research. A notebook is an interactive application that combines live code (read-print-eval loop or REPL), narrative, equations and visualizations. Common notebooks use an interpreted languages such as Python or R, and narrative follows markdown syntax. Data analysis greatly benefits from using notebooks instead of plain text editors or even IDEs: the combination of visuals and texts allows researcher to tell compelling stories about their data, and the interactivity of its code enables quick testing of different strategies. Jupyter notebook is a popular web-based interactive notebook developed originally for Python coding, but also accepts R and other programming languages upon installation of their kernels—a computing engine that executes the notebook's live code "under the hood". Jupyter notebook can also be run in the cloud using platforms such as Google CoLab and Amazon WebServices, taking advantage of the current trend of cloud computing. RStudio also allows the generation of R-based notebooks known as R Markdown, which is specially well-suited for generating reports.

Step 4: Follow good coding practices

After dealing with steps one to three, finally comes writing code. Coding, however, requires good practices to ensure correctness, sustainability and reproducibility for you, your future self, your collaborators (as we will discuss in Level 2) and the whole community (as we will discuss in Level 3). First and foremost, you need to make sure your code works correctly. In computational biology, correctness implies biological and statistical soundness. Both are big topics beyond the scope of this

manuscript. To achieve the former, however, a useful approach is to design positive and negative controls in your program, analysis or workflow. In an experiment, a positive control is a control group that is expected to produce results; a negative control is expected to produce no results. The same approach can be applied to computation, using input data whose output is previously known. Biological soundness can also be tested by quickly assessing expected orders of magnitude in both, intermediate and final files.

Beyond the correct functioning of your code, you will need to pay attention to the way your code looks, also know as "coding style". This includes a series of small and ubiquitous decisions regarding where and how to add comments; indentation and white spaces usage; variable, function and class naming; and overall code organization. It is true that, as in writing, there is a lot of your own personality in the way you code. However, sticking to existing coding style rules facilitates collaborations with your future self and others. Indeed, as we sometimes have trouble reading our own handwriting, we can also struggle reading our own code if we overlook any guidelines. At the very least, your code must display internal consistency. Even better, you can follow any of the multiple coding style rules that have been published. Although arbitrary, most of these rules have been developed with readability in mind. A good place to start, however, are style guides from software development teams. Google, for example, has published guidelines for Python, R, Shell, C++, and HTML/CSS [17]. Also, a series of guidelines for the Python programming language have been published with the name of Python Enhancement Proposal (PEP), where the most widely adopted is PEP-8 [18]. To aid flagging stylistic errors in your code, tools called "linters" are usually included with code editors and IDEs or provided as plugins.

In the matter of code styling, two topics merit additional attention: variable naming and comments. Variable names should be descriptive enough to convey an idea about the variable, function or class' content and use. The goal is to produce "self-documented" code that reads close to plain English. To do so, use multi-words variable names if necessary. In such cases, the most common conventions include Camel Case, where the second and subsequents words are capitalized ("camelCase"); Pascal Case, where all words are capitalized ("PascalCase"); and Snake Case, where words are separated by underscores ("snake_case"). Notably, all these conventions can be used in a same coding style to differentiate variables, functions and classes. For example, PEP-8 recommends Snake Case for functions and variables, and Pascal Case for class names. In addition to master variable naming, code comments—explanatory human-readable statements not evaluated by the program—are necessary to enhance the code's readability. No matter how beautiful and well-organized your code is, high-level code decisions will not be obvious unless stated. As a corollary, code explanations that can be deduced from the syntax itself should be omitted. Comments can span a single line or several ones, forming a block, and can be found in three strategic parts: at the top of the program file ("header comment"), which contains the author and the date of the code and what it accomplishes; above every function ("function header"), which contains the purpose and behavior of the function; and in line, next to tricky code whose behavior is not obvious or warrant a remark.

When working with a sizable code base, a good practice related is to strive for modularity—splitting your code's functionalities into independent entities known as modules. Modularity enhances code readability and reusability—enabling your code to be used by you or others in future applications—and expedites maintenance. In Python, subdivisions are defined as follow: a module is a collection of functions and global variables, a package a collection of modules, a library a collection of packages, and a framework a collection of libraries. Modules are simply files with the `.py` extension. Packages, in the other hand, must be indicated to the Python interpreter adding a file named `__init__.py` (which could be empty or not).

Code styling rules also apply to data science notebooks. However, when writing notebooks you must also engage in "literate programming"—a programming paradigm where the code is accompanied by human-readable explanation of its logic and purpose. In other words, notebooks must tell a story

about the analysis, connecting the dots between the code, the results and the figures. Human-readable language is often written in Markdown—a lightweight markup language. Little has been written about good practices for literate programming, but we advice you to explain the purpose of each chunk of code and provide some interpretation of its results.

Equally as important as writing good code is to use version control—the practice of tracking and managing changes in your code. This is a good and necessary practice even if your only collaborator is your future self. One of the main advantages of version control is keeping a change log of your files that can be utilized to go back to previous versions of your code, and remind you of previous approaches disregarded in newer versions. The most widely used version control system, Git, achieves these tasks with the command `git checkout`. Additionally, version control also allows you to safely try new functionalities using "branches"—carbon copies of the main original branch (known as "master") where you can add code independently and optionally merge it to the original one. Git creates branches using the command `git branch` and the same `git checkout` can be used to switch among them. We will discuss the utility and implementation of branches in collaborative projects in the next section (See Level 2: Collaboration). Nowadays, there are multiple code repositories that also provide version control with Git, such as GitHub, GitLab or Bitbucket. They have the additional benefit of backing up your code and code history in the cloud, keeping your work safe and shareable. You can interact with these platforms using the browser or via graphic user interfaces (GUI) such as GitHub Desktop or GitKraken.

Table 1: Steps to start any computational biology project.

Step	Use case	Common tools
Step 1: Choose your programming languages	Interacting with a Unix/Linux HPC	• Shell/Bash
	Data analysis	• Python, R
	Scripts and programs:	• Interpreted: Python, R, Perl • Compiled: C/C++, Rust
	Workflows	• Snakemake (Python), Nextflow (Groovy), CWL
Step 2: Define your project structure	Project structure	• Templates: Cookiecutter Data Science, rr-init • Workflows: Snakemake structure
	Virtual environment managers	• Language-specific: pipenv (Python), virtualenv (Python), renv (R) • Language agnostic: Conda
	Package managers	• Language-specific: pip (Python),BiocManager (R), R Studio package manager (R) • Language-agnostic: Conda
Step 3: Choosing your working set-up	Text editors	• Desktop applications: Atom, Sublime, Visual Studio Code, Notepad++ • Command line: Vim, Emacs
	IDEs	• For Python: Jupyter Lab, JetBrains/PyCharm, Spyder • For R: R Studio
	Notebooks	• Jupyter (Python, R), R Markdown (R)
Step 4: Follow good coding practices	Coding style	• Styling guides: PEP-8 (Python), Google (Python, R) • Linters
	Literate programming	• Markdown

Step	Use case	Common tools
	Version control	• Version control system: Git • Code repositories: GitHub, GitLab, Bitbucket • Git GUIs: GitHub Desktop, GitKraken

Level 2: Collaboration

Collaboration is a key aspect of scientific research, but it is especially relevant in computational biology, where interdisciplinary knowledge is often needed. Collaborators can take different forms: your boss or advisor, colleagues or lab mates, other laboratories, people from academia or industry, or your future self (as discussed in Level 1). Although collaborators can have a wide range of involvement with your project—from co-authors to commenters—they all share a direct relationship with you and your research, comprising a group of a dozen of people at most (contrary to a community, which is an open group of a large number of people, as we will discuss in Level 3). Each type of collaboration requires its own set of good practices, which we will cover in the next paragraphs.

2.1 Share code

Sharing code is one of the most common practices in software development, where large teams work together developing highly complex functions and scripts. Although computational biology projects are usually not as big, proper ways of sharing code are still essential, as it is not desired to have file conflicts as soon as two different researchers change the same piece of code. As mentioned in the previous section, hosting services such as GitHub [19], GitLab [20] and Bitbucket [21] (Table ??) allow for having a Git repository stored online, by creating a copy of the repository known as the "remote", which becomes the official version of the repository. The key advantage of using a remote is that there will be no direct interaction between different local copies of the repository, also known as a "clones", but instead each clone only will interact with the remote, and can only update the remote if there are no conflicts. This way, if a collaborator updated the repository with some changes, another collaborator will not be able to send their changes until they make sure to update their local copy with the changes already present online.

Tools for collaborative research. {#tbl:collaboration-tools}

Goal	Tools
Share code	• *Hosting services*: **GitHub** [19], **GitLab** [20], **Bitbucket** [21]. • *Git branching strategies:* **Github flow** [22]. • *Tests:* correctness (e.g. **pytest** [23]), coverage (e.g. **codecov** [24]), automation (e.g. **tox** [25], **Travis CI** [26], **Github Actions** [27]). • *Code reviews*: **Github** [28], **Crucible** [29], **Upsource** [30].
Share data	• *FAIR principles* [31]. • *Tidy data* [32]. • *Data version control* [33].
Share data science notebooks	• *Static:* **GitHub**, **GitLab**, **NBviewer** [34]. • *Interactive:* **Binder** [35], **Google CoLab** [36]. • *Comparative:* **ReviewNB** [37].
Share workflows	• *General hosting services*: **GitHub**, **GitLab**, **Bitbucket** • *Dedicated workflow repositories:* **WorkflowHub** [38]

Goal	Tools
Share manuscripts	• *General-purpose word processors:* **Google Docs** [39], **Office 365** [40]. • *Scholarly word processors:* **Authorea** [41]. • *Online applications supporting Markup Languages:* **Overleaf** (LaTeX) [42], **Manubot** (Markdown + GitHub) [43].

In order to guarantee that different collaborators can work simultaneously in the same repository, a good idea is to implement some type of branching strategy in the repository (Table ??). In a small team of collaborators, the most common strategy is to have a single `master` branch and generate from it branches that each different developer can work on. Then, whenever the developer is ready, they can request to combine, or "merge", the changes from their branch into the master branch, in a process known as "pull request", or PR for short. Once a PR has been opened, collaborators can review it, and if it fulfills their criteria, approve it so that it can be merged into the master branch: any succeeding branches will now have those commits already included as part of their history. This branching strategy is sometimes referred to as Github flow [22] and will suffice for most projects. For more complex branching systems, see Section 3.3: Make your research sustainable.

Using Git hosting services for collaboration has many additional benefits. The commit history not only shows what was done at each point in time, but also which collaborator did it, so that if e.g. a bug was introduced, commands such as `git blame` will show which collaborator caused it. Collaborators can also create "forks", i.e. full copies of repositories under their own possession, for e.g. having different a version of the software that works for a different purpose. Git hosting services can be accessed interactively online, or from the terminal with tools such as GitHub CLI [44]. Finally, Git hosting services also allow collaborators to open issues [45] for listing pending to-do's and/or asking questions, acting as an open forum for development discussions, which has the advantage of remaining accessible for the future, if new collaborators would join the research project later (as opposed to closed e-mail discussions). We will discuss additional advantages of using Git hosting services, in terms of interacting with a user base, in Level 3: Community.

Another important concept to internalize when developing code, especially together with other collaborators, is to develop unit tests (Table ??). Unit tests are scripts that will run to determine if specific modules/functions work as intended within the codebase, so that if later the function grows in scope, its proper basic functioning is ensured. For instance, if a function was defined for adding numbers, a simple test would be to asses if the function outputs 13 when the inputs 6 and 7 are provided. Tools such as `pytest` [23] for Python and `testthat` [46] for R exist to then detect said scripts, and run all of them to display if any specific section is failing. It is a very good practice to develop tests at the same time you develop code (at the personal research level), as adding tests *a posteriori* is significantly harder (but sometimes inevitable). Going beyond testing correctness, tools such as `flake8` [47] will test styling preferences (for complying with PEP8), `safety` [48] will test for vulnerabilities among the software's dependencies, and Codecov [24] will test which percentage of the codebase is tested, given that as a rule of thumb, the more code lines tested, the more reliable a software is. All these different types of tests can be funneled into a single testing pipeline that can run automatically whenever desired. This process is known as Continuous Integration (CI), and can be tuned to run locally whenever commits are made, or online whenever a pull request is opened and/or merged. When running locally, an environment manager / command line tool such as `tox` [25] helps to ensure all tests are ran under e.g. different python versions. For setting up the CI cycle online, different dedicated CI tools such as Travis CI [26] or Circle CI [49] exist, and more recently GitHub actions [27] has been introduced for running the integration directly from Github.

Having tests is a great way of ensuring code fulfills a certain level of correctness and styling. However, it is no replacement for a human assessment to see if the code is correct, necessary and useful. Therefore, code reviewing is essential whenever developing code in collaboration (Table ??). Tools such as Crucible [29] and Upsource [30] exist for making in-line reviews of each file, but the most

common approach, if the software is stored in a repository, is to directly review using the online review tools from the hosting service. In the case of Github [28], this not only allows the reviewer to open a comment in any line of the code (which creates a thread for the original author to reply in), but also to suggest changes, for the author to in turn approve/dismiss. When reviewing, there are a series of things to look for (from functionality to documentation), and good practices to keep in mind (such as phrasing the comments in a constructive way), which are outside of the scope of this review but presented in detail elsewhere [50,51].

2.2 Share data

The practices of sharing data stem from the same place as with sharing code: we should store our dataset and any changes to it in a repository, and ensure it complies with standards by testing its quality. However, due to data having a more consistent structure than code, as data is often processed and outputted by machines in standard formats, there are additional criteria that should be considered when we share it with collaborators (and later on with the community). The main set of guidelines that represent these criteria were outlined a few years ago in what is known as the FAIR principles [31]: data should be Findable, i.e. easy to find online; Accessible, i.e. easy to access once found; Interoperable, i.e. easy to integrate with other data/applications/workflows/etc.; and Reusable, i.e. presented in a way that allows for others to use it for the same purpose or different settings.

For making data findable, research repositories such as Zenodo [52] and Figshare [53] allow you to assign a digital object identifier (DOI) to any group of files you upload, including data and/or code. Alternatively, regular code repositories like Github can be used instead, as you can use specific commits and/or releases to identify specific versions of the data (see 3.1. Make your research accessible), in combination with extensions for Large File Storage such as git LFS [54], in the case of data files larger than 100 MB [55]. A final alternative is the Data Version Control (DVC) initiative [33], which is especially useful when doing machine learning, as it can keep track of data, machine learning models and even scoring metrics.

For making data accessible, we encourage as much as possible to make your repositories open access so that everything is accessible to everyone, but in cases in which you or your collaborators prefer some restrictions, you can create guest accounts to provide access to private repositories. For making data interoperable, distinctions between raw and clean data have been made [**???**], with the raw data being the same files that came out of the measuring device, and the clean data the files that are ready to be used for any computational analysis. An important characteristic that clean data should have is to be “tidy”, which is reviewed in detail elsewhere [32]. Finally, for making data reusable, thorough documentation of the data - including experimental design, measurements units and sources of error - is required.

2.3 Share data science notebooks

As we previously discussed, Jupyter Notebook have become a fundamental tool of data analytics. Accordingly, you will likely need to share your notebook with collaborators at some point. To do this, there are static and interactive options. The former, as the name indicates, share computational notebooks as a rendered text, written internally in HTML. Static notebooks are a good option when you want to avoid any modifications and can work as an archive of past analyses, but interacting with its content is cumbersome—the file must be downloaded and run in a local Jupyter installation. Git-based code repositories, such as GitHub and GitLab, automatically render notebooks that can be later shared pointing collaborators to the GitHub repository. To ease this process, the Project Jupyter provides a web application called NBviewer, where you can paste a Jupyter Notebook’s URL, publicly hosted in GitHub or elsewhere, and renders the file into a static HTML web page with a stable link.

Interactive notebooks, in the other hand, not only render the file but also allow collaborators to fully interact with it, tinkering parameters or trying new input data—no installation required. The Binder Projects (which is also part of the Project Jupyter) offers the Binder service, where any publicly hosted Git-based repository can be open with a Jupyter Notebook interface. The user can fully interact with any notebook within the repository, although changes will not be saved to the original file. The platforms supports Python and R among other languages, and any additional packages required to run the analysis need to be specified in a configuration file within the repository. Similarly, Jupyter Notebooks can be run interactively using Google CoLaboratory (CoLab), which is available to anyone with a Google account. Notebooks can be updated locally, from any public GitHub repository, or from Google Drive, where also imported files are also saved. In both cases, the machines provided by these services are comparable to a modern laptop. Thus, these tools are not suitable for some computing-intensive tasks common to computational biology problems.

Besides sharing notebooks, oftentimes computational biologists need to work and edit a notebook together. In those cases, notebooks need to be treated as any other piece of code: updates from different collaborators must be managed with version control in a platform such as GitHub. The problem, however, is that Git-based hosting services deal with notebooks as if they were HTML text, where changes between versions are hard to visualize. To better compare these changes, there is NBreview, which renders and display in parallel the old and new versions of a notebook for easy comparison. The tool can be easily installed using your GitHub account, and notebooks can be reviewed from their website.

2.4 Share computational workflows

Computational biology projects often demands sharing multi-step analyses with dozens of third-party software and dependencies. Although these steps can be shared as documentation, complex workflows are better shared as stand-alone code that can be easily run with minimal file manipulation from collaborators. Doing so can safeguard the reproducibility and replicability of the analysis, leading to better science and less issues down the road.

The simplest way to share a pipeline is to generate a Bash script that receives input files from the command line, thus, allowing to run it with different input data. However, Bash scripts offer little control over the overall workflow and cannot re-run specific parts of the pipeline. To address these issues, pipelines are better shared using a workflow automation system. Theoretically, all the instructions regarding the workflow could be written in the main pipeline file—in Snakemake, the `.smk` file (or Snakefile); in Nextflow, te `.nf` file; and in CWL, `.cwl` file. However, to ensure reproducibility, it is a good practice to share complete pipelines, meaning folder structures, additional files and software specification, as well all custom scripts developed for the analysis. These files can be shared using the same tools as other forms of code, namely GitHub or any other Git hosting services. Alternatively, they can be uploaded to hosting services specialized in workflows, like WorkflowHub [38], currently in beta.

When sharing workflows, consider that sharing software versioning is necessary for your collaborators to reproduce your pipeline using their own computing setup. Conda environments, for example, can be easily created from an environment file (in YAML language), which can be exported from an existing environment. Notably, Snakemake and Nexflow can be configured to automatically build isolated environments for each rule or step, enabling running different versions of a program within the same pipeline (which is especially helpful when needing both Python 2 and 3). Besides sharing the specifications of an environment, it is possible to share the environment itself via containers, using platforms like Docker and Singularity, which are especially helpful to share environments with a broader community, as we will discuss in Level 3.

2.5 Write manuscripts collaboratively

Writing articles is arguably the main way where we can share our research with the scientific community and for that purpose, the world. However, in a highly interdisciplinary field as computational biology, writing manuscripts is also a collaborative effort, where multiple people is directly involved in the crafting of a manuscripts. The traditional computer tools for writing documents, therefore, are oftentimes not suitable for this type of collaboration, resulting in files with different names jumping from one e-mail inbox to another, resulting in multiple and likely contradictory final versions. Let's avoid this by streamline collaborative manuscript writing with tools made for that purpose.

Big companies have become aware of the need for collaborative writing, developing online applications that can be simultaneously edited by multiple people. Google's GDocs and Microsoft's Office 365 are well-known word processors designed for this purpose, where the text is displayed with the exact appearance than in a printout (know as *What-You-See-Is-What-You-Get* , or WYSIWYG) and the text can be formatted making use of the internal features of the application. The advantage of these technologies is that they are extremely user-friendly, and require no additional knowledge. They are a good option when one or more of your collaborators seeks simplicity, but they are not specifically tailored for the needs of scientific writing, such as adding references, equations and figures. Fortunately, third-party companies have developed plugins for these applications to add references to your document. Companies like Authorea have developed their own online application specifically designed for writing manuscripts. Authorea, in particular, offers templates for different type of research projects, allows you to manage collaborators from the same platform, and easily add reference from using identifiers (DOI, PubMed, etc.). Consider, nonetheless, that some collaborators may not want to adopt a new tool exclusively for writing manuscripts.

In addition to word editors, text editors are a competitive option to write manuscripts when combined with a markup language—a human-readable computer language that uses tags to delineate formatting elements in a document that will be later rendered. Since the formatting process is internally handled by the application, styling elements (headers, text formatting, equations) can be easily written in text, even achieving greater consistency than word processors. Disciplines closely related to computational biology, such as statistics and mathematics, have historically used the markup language LaTeX for writing articles. This language has a specific syntax to write mathematics constructs as simple text, making it a sound choice for papers with lots of equations. To aid collaborative writing, platforms like Overleaf provide online LaTeX editors, supporting features like real-time editing. In addition to LaTeX, an emerging trend in collaborative writing is to use the lightweight markup language Markdown within the GitHub infrastructure. The software Manubot provides a set of functionalities to write scholarly articles within a GitHub repository, leveraging all the advantages of Git version control and the GitHub hosting platform [56]. For example, it provides cloud storage, version control, and facilitates the maintainers' work by managing updates via pull requests. The GitHub user interface also allows off-line discussions about the manuscript using issues" and task assignment (See level 3 for tips on project management). Manubot, in particular, accepts citations using manuscripts identifiers and renders automatically renders the article in PDF and HTML formats. As a drawback, it requires technical expertise in Git and familiarity with GitHub; as an upside, its reliable infrastructure scales well to large and open collaborative projects.

Level 3: Community

The third and final step of this journey is when you want to present your research to the community. This can already be done shortly after the start of your project, or towards the end, when you are close to publish your results. Either way, the main goal when doing it should be to develop and maintain an open and reproducible computational biology project, with community engagement over

time. Here, we can distinguish 3 sub-goals: Make your research 1) accessible, 2) reproducible and 3) sustainable. The latter is especially relevant when part of the research involves developing code that will be used by others in the future (e.g. a tool or workflow), but we believe that many of those ideas are still relevant to any computational biology project as well.

3.1. Make your research accessible

Making your research accessible is the first step towards open research. This include ensuring that anyone (inside or outside of academia) can access your research, and that your research stays available long after your paper is published. It is extremely frustrating for any researcher to look for software or a set of scripts from a paper published a few years ago, only to find a "404 error" where the code used to be provided. Even more frustrating is whenever authors offer code as "available on request"; this often leads to an email from the supervisor of the project explaining that the developer of the code left years ago so the code is not available anymore.

There are three main ways in which people publish accompanying code to publications: As supplementary material in the publication it was used, via privately-owned domains, or via public repositories. The first option (publishing the code as supplementary material) has low accessibility if the paper is not open access, as only members of institutions that pay the journal membership will be able to access it. Moreover, it is completely static and cannot ever be updated if a new feature wants to be introduced, or a bug is found. The second option (privately-owned domains) lacks sustainability, as it requires certain maintenance, and either the domain can expire or the website might migrate somewhere else. Therefore, here we argue for the third option (public repositories) because of its accessibility and sustainability over time, as owners can update the code (if necessary) in a git-compliant way, and they don't need to rely on maintaining a domain and/or servers by themselves. As mentioned before (see Level 2: Collaboration), there are several hosting services for this purpose [19,20,21] (Table 1), all equally valid depending on where people in your specific field usually publish their tools. As the authors have more experience with Github [19], in the following several repository-specific tools will be exemplified with Github.

Table 1: Tools for making your research accessible. For each highlighted goal, different tools that achieve that goal, together with additional remarks, are shown.

Goal	Tool options	Additional remarks
Publish your code	• **Github** [19] • **Gitlab** [20] • **Bitbucket** [21]	All three options allow you to host your repository online for free. Choose whichever is more common in your own field.
Introduce your code	• **README file** [57]: First file that shows up in a repository. • **Github Pages** [58]: Separate website.	Provide a landing page to any repository with a short overview of the code (installation, usage, acknowledgments, etc).
Share your code	• Several licensing options [59].	Indicate with a license file what restrictions apply when using your code. If you don't include this, you will loose many users.
Archive your code	• **Github Releases** [60] • **Zenodo** [52]: Provides DOI. • **figshare** [53]: Provides DOI.	Share progressive stable versions of your code as you develop it. Use **semantic versioning** [61] for assigning standard identifiers to your releases.
Publish a tool	• **PyPI** [62]: Python. • **CRAN** [63]: R. • **Bioconductor** [64]: R. • **Bioconda** [65]: Language-agnostic.	Produce a package easy to install and use. Especially useful if you think you could have a userbase that will run the same analysis as you on other datasets and/or conditions.
Publish an interactive web app	• **Dash** [66]: Python. • **R-Shiny** [67]: R.	Provide easy and interactive data exploration to your users. Especially useful if you have large datasets that can be explored in different ways.

When publishing any piece of code online, there are two files that are fundamental to include: A readme file, and a license. Including a readme file [57] is about welcoming your users, and introducing them to your code (Table 1): it should include a description about its main intended use, an overview of the installation, the most common commands, a way to contact the developers if problems arise (see 3.3. Make your research sustainable for more details on this), and acknoledgments (if appropiate). We recommend to keep it short: You can include all the details in the documentation of the tool, here we only need a quick overview. The readme file, which can be written in different markup languages such as Markdown [68] or reStructuredText [69], will render automatically on the repository's landing page, below the repository file structure. For improved clarity, you may consider creating a separate landing page: it will declutter the look by only showing the information of the readme file, and use improved esthetics. All hosting services offer simple ways to do this; in the case of Github, you can use Github pages [58].

Adding a license to a repository is also a crucial step (Table 1). Licenses indicate how the code can be used: Is it free to use for any application? Can users modify the code as they please? Does it come with a warranty that it will work? Can it be used for profit? If a license is not provided, many researchers will choose not to use the code at all, as they won't have an answer to these questions. For instance, academic users will not know if they are entitled to modify the code for their own research, and users from the industry will not know if the code can be used for profit. Many options exist for licensing code [59], from permissive licenses that allow any kind of use with few or no conditions, like the Unlicense and MIT licenses, to more restrictive licenses that enforce disclosing the source and even requiring that any adaptation of the code uses the same license, like the GNU licenses. Our suggestion is to choose whatever suits your research group best; consider that as a rule of thumb, the more requirements you add, the less potential users you will have, but the more credit you will receive when users utilize your research for their own needs.

Working as a computational biologist, you will probably continue lines of work from scripts or software you have already been published. For instance, perhaps you improve performance of a given function, or add a new set of features entirely. Therefore, not only you should be interested in your code being accessible as a single instance, but instead provide ways of accessing numerous versions. To keep different versions of your code organized as you develop it, it is important to create and archive succeeding releases of your code (Table 1). A good way to do this is with Github releases [60], which requires minimal effort and shows all versions you have created of your code. An even better way to do this is with research repositories like Zenodo [52] or Figshare [53], which not only store your code and data, but also give you a digital object identifier (DOI) for each version, making different versions of your code citable, in case the corresponding publication is not publicly available yet, or enough time has passed since the publication so that the current version of the code differs widely from what was published. These repositories can even be combined with code repositories, e.g. Github has a Zenodo integration that will trigger a new archived version every time a new release is made from Github. Regardless of the solution chosen, we recommend keeping some logical order to the releases, using a standard such as semantic versioning [61].

In most cases, it is probably enough to provide your code as an organized set of scripts and/or notebooks, for anyone to consult if they wish to reproduce and/or re-utilize your code. However, if you believe that your code would be partly or entirely used in a routine-fashion by other researchers, for instance for studying other organisms or other experimental conditions, you could consider packaging your code as a tool (Table 1). If you take this route, aim for using one of the main options for publishing tools as packages: Bioconda [65] is a catch-all solution, but language-specific options also exist, such as PyPI [62] if you work in Python, or CRAN [63] and Bioconductor [64] if you work in R. All of these options will allow you to reach a bigger audience, as packages can be easily searched and installed locally with minimal effort.

Another common situation in research projects is, instead of having code that can be used by others to analyze their data, having data that can be analyzed in many ways. As producing plots that display the data in every conceivable way is unreasonable, a good solution here is to develop an interactive web app (Table 1), also referred to as a data dashboard, that lets users interact with the data, by showing different sets of variables or changing parameter settings (e.g. the significance of a statistical test). Common options for this goal are Dash [66] for Python and R Shiny [67] for R.

3.2. Make your research reproducible

Having your code/data accessible to anyone is only the first step when sharing software to the research community; you also need for any analysis you have conducted to be reproducible by anyone. We have already discussed the importance of reproducibility in science when working in a personal project (see Level 1: Personal Research); however, when sharing your results with the academic community this becomes even more paramount, as anyone should be able to run your code, and obtain the same results. This is especially relevant in science, as there is a constant number of people that are new to the field (including undergraduates, postgraduates and senior researchers), many who come from very different backgrounds to try to understand how your code, developed for your own specific niche research area, works.

A cornerstone for reproducibility is to have a solid documentation: by explaining what your code does, users can understand the intended use of each function, which will guide them on how to achieve your same results. We can distinguish 4 different levels of documentation [70]:

- *Tutorials:* A group of lessons that teach the reader how to become a user of your code.
- *How-to guides:* A set of documents that clarify to a user how to solve common problems/tasks.
- *Explanations:* Discussions that clarify particular topics related to your code.
- *References:* Technical descriptions of your code's variables/classes/functions.

The extent of documentation you write will depend on how many users you expect to have, and conversely will affect how many users you attract. If you foresee that your code has little usability outside of your research, perhaps a solid documentation of each function using docstrings [71] could already be enough. However, you might also want to add a tutorial for a beginner, a couple of how-to guides for frequently used routines, and even some explanations for clarifying the science behind your code. The latter has the added bonus that it can be re-used for the eventual manuscript of your publication. If you are not sure how many users you might get, air on the side of caution and prepare good documentation anyways: You will be surprised to see how often other researchers contact you for reproducing your results and/or using your code for other applications! Finally, to publish comprehensive documentation online, consider using 1) a standard documentation language such as reStructuredText [69] or Markdown [68], and 2) a documentation platform such as Readthedocs [72] or Gitbook [73] (Table 2).

Table 2: Tools for making your research reproducible. For each highlighted goal, different tools that achieve that goal, together with additional remarks, are shown.

Goal	Tool options	Additional remarks
Document your code	• **Readthedocs** [72]: Uses **reStructuredText** [69]. • **Gitbook** [73]: Uses **Markdown** [68].	Comprehensive documentation: from tutorials and how-to guides all the way down to function documentation based on all compiled **docstrings** [71].
Reproducible environments	• **Environment managers**: See Table XXX. • **pip-tools** [74]: Administer several environments in a single project.	As a recommendation, try having the minimum number of dependencies needed to reproduce your results.

Goal	Tool options	Additional remarks
Reproducible software	• **Docker** [75] • **Singularity** [76]	Package your research as a container ready to run in any computer.
Reproducible commands	• **Make** [77]	Build a program by following a series of steps in a single Makefile.
Reproducible workflows	• **Make** [77] • **Snakemake** [78]: Uses a Python-based language. • **Nextflow** [79]: Uses **Groovy** [80].	Run a pipeline of commands on NGS data in a reproducible way.
Reproducible notebooks	• **Binder** [35] • **Colaboratory** [36]	Make your notebooks interactive and reproducible.

TODO: write about environments -> depends on what is written in Level 1.

A tool worth using when defining each environment is pip-tools [74], which allows defining several different environments for a single project, depending on who accesses the project. For instance, you might want users to use a wider number of dependencies for increased flexibility, but the CI tool to only have a minimal number of dependencies for improved efficiency.

Beyond dependency trackers, you might also want to ensure your tool behaves in the same way across computing environments, even between two different operative systems (e.g. Mac and Windows). The best solution for this is to use a container (Table 2), which is a standardized unit of software that runs using an isolated filesystem known as a container image. This image contains not only the needed dependencies, but also any configurations, binary files, environmental variables, etc. that the software needs for running, and can be built from a text file with all the instructions. The two main tools available for creating containers for free are Docker [75] and Singularity [76].

TODO: make, as any type of workflow -> depends on what is written in Level 2.

There are certain research projects where particular resources for ensuring reproducibility are especially useful: as previously mentioned (see Level 2: Collaboration) in the case you have developed a complex workflow, for example a pipeline for analyzing NGS data, tools like Snakemake [78] (which uses a Python-based language) or Nextflow [79] (which uses **Groovy** [80]) will certainly be helpful. Finally, in the case that you have prepared most of your code using Jupyter notebooks, tools like Binder [35] or Colaboratory [36] allow you to publish interactive reproducible notebooks that anyone can run from their own setup.

3.3. Make your research sustainable

Now that your research can be accessed and reproduced by anyone, the final step is to keep it like that over time. This is especially relevant if you carry the research further by integrating new features, as you need to ensure no bugs appear in the rest of the code, and at the same time offer options for your user community to request new functionalities, in order to foster a strong community over time. However, even in the case in which your research is a self-contained project that will not be continued, it is still important to ensure that your user community has ways to contact you, in case bugs would be discovered in your code or, due to packages/dependencies updating, parts of your code would not work anymore over time (a common phenomenon referred to as “software rot” [81]). In the following we will therefore review useful techniques for making your code/software/research sustainable over time.

The first thing to consider when ensuring sustainability is to offer the users of your code a way to communicate with you (Table 3). You can achieve this with different tools, depending on the size of

your userbase and the scope of the questions you receive. For smaller projects, where there are only questions once in a while, a single-channel solution like Gitter [82] is probably enough, as it offers a simple way for anyone in the community to ask questions, and any of the developers to answer in threads. For larger projects however, it could become unmanageable to have all discussions in the same channel, so multiple-channel solutions, i.e. forums, are better suited. Google groups [83] is a good example that works well for large pieces of software, as it allows for anyone to open separate threads for different issues. Recently, Github introduced Github Discussions [84], also meant for keeping questions organized in different threads.

Table 3: Tools for making your research sustainable. For each highlighted goal, different tools that achieve that goal, together with additional remarks, are shown.

Goal	Tool options	Additional remarks
Tell users how to contact you	• Specific/shorter questions: **Gitter** [82]. • Larger issues / how-to's: **Google groups** [83], **GitHub Discussions** [84].	Provide ways for users to contact you for questions, requests, etc. Remember to visit them periodically!
Track to-do's in your research	• **Github Issues** [85]	Detail specific pending to-do's in your research / allow others to request changes and/or highlight bugs.
Encourage user contributions	• **Contribution guidelines** [86]: How to open issues / contribute code. • **Github Wikis** [87]: More specific how-to guides.	Provide as much information as you can to guide your users. You can also include administrator guidelines.
Foster a respectful community	• Smaller projects: **Contributor Covenant** [88]. • Larger projects: **Citizen Code of Conduct** [89].	Essential when you would like researchers to contribute code.
Branch your repo sustainably	• **Gitflow** [90]	Useful when several developers contribute code to the project. Allows users to get access to stable versions of your research in an ongoing project.
Keep track of your issues	• **Kanban flowcharts** [91]: **Github Projects** [92], **GitKraken Boards** [93]. • **Scrum practices** [94]: **Zenhub** [95], **Jira** [96].	Keep track of your pending tasks in different projects with **Agile** [97] software development practices. Especially useful if your research is split in many different repositories, each with multiple features/fixes to do.
Automate your repo	• **bump2version** [98]: Easier releasing. • **Danger-CI** [99]: Easier reviewing.	Do less, script more!

TODO: Issues (GitHub issues/milestones/labels/templates): Have a to-do list -> based on what is written in Level 2

Now that your users know *where* to contact you, you should also tell them *how* to contact you. For this, it is essential to have a file with contribution guidelines [86] (Table 3), detailing how should users 1) open issues and 2) contribute with their own code changes, in form of pull requests. These guidelines are mainly intended for new users/contributors, so they should be written in the style of a *how-to* guide. However, they may also include additional instructions for the main developers, or even the administrator of the repository. Alternatively, those more detailed guidelines can be included in a supplemental wiki, which hosting services offer as part of the repository [87].

Equally important to the contribution guidelines file is the code of conduct file (Table 3), which includes guidelines on how to behave when engaging with the community in the repository, and what to do if someone does not comply with these guidelines. It is an important statement to ensure users

and developers respect one another. Several templates exist as code of conduct, such as the Contributor Covenant [88] for smaller projects, and the Citizen Code of Conduct [89] for larger projects.

Finally, ensuring sustainability of your project means not only allowing and encouraging users contacting you, but also smarter ways to develop and maintain your software as it grows in scope and number of users. This includes:

1. *Branching System:* When many developers are involved in a project, you want to ensure that users can access functional versions of your code while you work on it. In this case, more advanced branching methods such as GitFlow [90] might be preferable (Table 3). With GitFlow, an additional development branch (often named as `devel`) is used as main branch for new branches to be based on, leaving the master branch as a separate branch only for stable versions of the code, and a merge from development to master always invoking a code release. This way, users can directly access the master branch for tested releases, whereas the latest additions of the code, perhaps not 100% tested, will be available for developers to further work on in the development branch. Additional branches can be added depending on the scope of the project, such as specific version branches in case further testing is needed, or hotfix branches, in case bugs are detected in master and a quick solution is required.

2. *Project Management:* Although issues are fundamental for keeping track of what is there to do, they can become hard to organize and prioritize as they grow in number. Several project management tools exist to solve this (Table 3), all based on Agile [97] principles. The most simple one is to have a Kanban board [91], where issues are organized in as many columns as necessary to have a clear layout and figure out what is the current state of a given task. Tools like Github Projects [92] or GitKraken Boards [93] use this approach. For Larger projects with either several collaborators and/or several repositories, a more structured approach such as a Scrum framework [94] might be needed, as issues are more easily prioritized by setting milestones, estimated by assigning points, and dependencies can be defined. Both Zenhub [95] and Jira [96] are great options for this.

3. *Additional Automation:* As you develop your project, you will find that many aspects can be automated to improve efficiency. Want to make releasing faster to ensure all sections of your code get updated with the new release? bump2version [98] might help. Want to make sure contributors comply with certain standards in their pull requests? Look into Danger-CI [99]. Note here that we advice against implementing all of this from the start, but instead adding different tools as you realize you need them. If you find yourself performing a task in a routinely fashion, ask yourself if you could automate it. More often than not the answer is yes!

TODO: More automation options?

Case studies

- Example of computational biology project 1: RNA-seq analysis (workflow)
- Example of computational biology project 2: Genome-scale metabolic model (systems biology project)
- Example of computational biology project 3: Software development (computational tool)

Conclusion

Pending

Acknowledgments

Pending

References

1. **NIH working definition of bioinformatics and computational biology**
   Huerta Michael, Downing Gregory, Haseltine Florence, Seto Belinda, Liu Yuan
   (2000-07-17)
   https://www.kennedykrieger.org/sites/default/files/library/documents/research/center-labs-cores/bioinformatics/bioinformatics-def.pdf

2. **What is bioinformatics? A proposed definition and overview of the field.**
   NM Luscombe, D Greenbaum, M Gerstein
   Methods of information in medicine (2001) https://www.ncbi.nlm.nih.gov/pubmed/11552348
   PMID: 11552348

3. **Good enough practices in scientific computing**
   Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, Tracy K. Teal
   PLOS Computational Biology (2017-06-22) https://doi.org/gbkbwp
   DOI: 10.1371/journal.pcbi.1005510 · PMID: 28640806 · PMCID: PMC5480810

4. **Software engineering for scientific big data analysis**
   Björn A Grüning, Samuel Lampa, Marc Vaudel, Daniel Blankenberg
   GigaScience (2019-05) https://doi.org/gf4f4m
   DOI: 10.1093/gigascience/giz054 · PMID: 31121028 · PMCID: PMC6532757

5. **So you want to be a computational biologist?**
   Nick Loman, Mick Watson
   Nature Biotechnology (2013-11-01) https://doi.org/p3j
   DOI: 10.1038/nbt.2740 · PMID: 24213777

6. **Ten Simple Rules for Taking Advantage of Git and GitHub**
   Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, … Juan Antonio Vizcaíno
   PLOS Computational Biology (2016-07-14) https://doi.org/gbrb39
   DOI: 10.1371/journal.pcbi.1004947 · PMID: 27415786 · PMCID: PMC4945047

7. **Ten Simple Rules for Reproducible Research in Jupyter Notebooks**
   Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, Peter W. Rose
   arXiv (2018-10-13) https://arxiv.org/abs/1810.08055v1

8. **Streamlining Data-Intensive Biology With Workflow Systems**
   Taylor Reiter, Phillip T. Brooks, Luiz Irber, Shannon E. K. Joslin, Charles M. Reid, Camille Scott, C. Titus Brown, N. Tessa Pierce
   Cold Spring Harbor Laboratory (2020-11-16) https://doi.org/gg353v
   DOI: 10.1101/2020.06.30.178673

9. **A Padawan Programmer's Guide to Developing Software Libraries**
   James T. Yurkovich, Benjamin J. Yurkovich, Andreas Dräger, Bernhard O. Palsson, Zachary A. King
   Cell Systems (2017-11) https://doi.org/gg8tqz
   DOI: 10.1016/j.cels.2017.08.003 · PMID: 28988801

10. **2018 Kaggle Machine Learning & Data Science Survey** https://kaggle.com/kaggle/kaggle-survey-2018

11. **Modulecounts** http://www.modulecounts.com/

12. **Why scientists are turning to Rust**
    Jeffrey M. Perkel
    Nature (2020-12-01) https://doi.org/ghqc7g
    DOI: 10.1038/d41586-020-03382-2 · PMID: 33262490

13. **Home - Cookiecutter Data Science** https://drivendata.github.io/cookiecutter-data-science/

14. **Reproducible-Science-Curriculum/rr-init**
    Reproducible Science Curriculum
    (2021-02-12) https://github.com/Reproducible-Science-Curriculum/rr-init

15. **A Quick Guide to Organizing Computational Biology Projects**
    William Stafford Noble
    PLoS Computational Biology (2009-07-31) https://doi.org/fbbpkn
    DOI: 10.1371/journal.pcbi.1000424 · PMID: 19649301 · PMCID: PMC2709440

16. **Distribution and Reproducibility — Snakemake 5.32.2 documentation**
    https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html

17. **google/styleguide**
    Google
    (2021-02-15) https://github.com/google/styleguide

18. **PEP 8 – Style Guide for Python Code**
    Python.org
    https://www.python.org/dev/peps/pep-0008/

19. **GitHub: Where the world builds software**
    GitHub
    https://github.com/

20. **DevOps Platform Delivered as a Single Application**
    GitLab
    https://about.gitlab.com/

21. **Bitbucket | The Git solution for professional teams**
    Atlassian
    Bitbucket https://bitbucket.org/product

22. **Understanding the GitHub flow · GitHub Guides** https://guides.github.com/introduction/flow/

23. **pytest: helps you write better programs — pytest documentation**
    https://docs.pytest.org/en/stable/

24. **Codecov - The Leading Code Coverage Solution**
    Codecov
    https://about.codecov.io/

25. **Welcome to the tox automation project — tox 3.21.5.dev3 documentation**
https://tox.readthedocs.io/en/latest/

26. **Travis CI - Test and Deploy with Confidence** https://travis-ci.com/

27. **Features • GitHub Actions**
GitHub
https://github.com/features/actions

28. **Reviewing changes in pull requests - GitHub Docs**
https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/reviewing-changes-in-pull-requests

29. **Crucible Code Review Tool for Git, SVN, Perforce and More**
Atlassian
Atlassian https://www.atlassian.com/software/crucible

30. **Upsource: Code Review and Project Analytics by JetBrains**
JetBrains
https://www.jetbrains.com/upsource/

31. **The FAIR Guiding Principles for scientific data management and stewardship**
Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, ... Barend Mons
Scientific Data (2016-03-15) https://doi.org/bdd4
DOI: 10.1038/sdata.2016.18 · PMID: 26978244 · PMCID: PMC4792175

32. **Tidy Data**
Hadley Wickham
Journal of Statistical Software (2014) https://doi.org/gdm3p7
DOI: 10.18637/jss.v059.i10

33. **Data Version Control · DVC** https://dvc.org/

34. **nbviewer** https://nbviewer.jupyter.org/

35. **The Binder Project** https://mybinder.org/

36. **Google Colaboratory** https://colab.research.google.com/

37. **GitHub Diffs & Commenting for Jupyter Notebooks** https://www.reviewnb.com/

38. **The WorkflowHub** https://workflowhub.eu/

39. **Google Docs: Free Online Documents for Personal Use** https://www.google.com/docs/about/

40. **Microsoft 365 with Office apps | Microsoft 365** https://www.microsoft.com/en-us/microsoft-365

41. **Open Research Collaboration and Publishing - Authorea** https://www.authorea.com/

42. **Overleaf, Online LaTeX Editor** https://www.overleaf.com

43. **Manubot - Manuscripts, open and automated** https://manubot.org

44. **GitHub CLI**
GitHub CLI
https://cli.github.com/

45. **About issues - GitHub Docs** https://docs.github.com/en/github/managing-your-work-on-github/about-issues

46. **Unit Testing for R** https://testthat.r-lib.org/

47. **Flake8: Your Tool For Style Guide Enforcement — flake8 3.8.4 documentation**
https://flake8.pycqa.org/en/latest/

48. **Safety - Security for your Python dependencies** https://pyup.io/safety/

49. **Continuous Integration and Delivery**
CircleCI
https://circleci.com/

50. **How to do a code review**
eng-practices
https://google.github.io/eng-practices/review/reviewer/

51. **Code Review Guidelines for Humans**
Philipp Hauer
Philipp Hauer's Blog (2018-07-31) https://phauer.com/2018/code-review-guidelines/

52. **Zenodo - Research. Shared.** https://zenodo.org/

53. **About**
figshare
https://figshare.com/about

54. **Git Large File Storage**
Git Large File Storage
https://git-lfs.github.com/

55. **What is my disk quota? - GitHub Docs** https://docs.github.com/en/github/managing-large-files/what-is-my-disk-quota

56. **Open collaborative writing with Manubot**
Daniel S. Himmelstein, Vincent Rubinetti, David R. Slochower, Dongbo Hu, Venkat S. Malladi, Casey S. Greene, Anthony Gitter
PLOS Computational Biology (2019-06-24) https://doi.org/c7np
DOI: 10.1371/journal.pcbi.1007128 · PMID: 31233491 · PMCID: PMC6611653

57. **Make a README**
Make a README
https://www.makeareadme.com

58. **GitHub Pages**
GitHub Pages

https://pages.github.com/

59. **Licenses**
    Choose a License
    https://choosealicense.com/licenses/

60. **Managing releases in a repository - GitHub Docs**
    https://docs.github.com/en/github/administering-a-repository/managing-releases-in-a-repository

61. **Semantic Versioning 2.0.0**
    Tom Preston-Werner
    Semantic Versioning https://semver.org/

62. **PyPI · The Python Package Index**
    PyPI
    https://pypi.org/

63. **The Comprehensive R Archive Network** https://cran.r-project.org/

64. **Bioconductor - Home** https://www.bioconductor.org/

65. **News — Bioconda documentation** https://bioconda.github.io/

66. **Dash Overview** /dash

67. **Shiny** https://shiny.rstudio.com/

68. **Daring Fireball: Markdown Syntax Documentation**
    https://daringfireball.net/projects/markdown/syntax

69. **reStructuredText** https://docutils.sourceforge.io/rst.html

70. **The documentation system — Documentation system documentation**
    https://documentation.divio.com/

71. **Python Docstrings**
    GeeksforGeeks
    (2017-06-01) https://www.geeksforgeeks.org/python-docstrings/

72. **Home | Read the Docs** https://readthedocs.org/

73. **GitBook - Document Everything!** https://gitbook.com/

74. **jazzband/pip-tools**
    Jazzband
    (2021-02-12) https://github.com/jazzband/pip-tools

75. **Empowering App Development for Developers | Docker** https://www.docker.com/

76. **Home**
    Sylabs.io
    https://sylabs.io/

77. **gnu.org** https://www.gnu.org/software/make/

78. **Snakemake — Snakemake 5.32.2 documentation** https://snakemake.readthedocs.io/en/stable/

79. **Nextflow - A DSL for parallel and scalable computational pipelines** https://www.nextflow.io/

80. **The Apache Groovy programming language** https://groovy-lang.org/

81. **What is Software Rot? - Software Development Glossary by DeepSource**
DeepSource
https://deepsource.io/glossary/software-rot/

82. **Gitter** https://gitter.im/

83. **Google Groups** https://groups.google.com/forum/m/

84. https://docs.github.com/en/free-pro-team

85. **Mastering Issues · GitHub Guides** https://guides.github.com/features/issues/

86. **Setting guidelines for repository contributors - GitHub Docs**
https://docs.github.com/en/github/building-a-strong-community/setting-guidelines-for-repository-contributors

87. **About wikis - GitHub Docs** https://docs.github.com/en/github/building-a-strong-community/about-wikis

88. **Contributor Covenant: A Code of Conduct for Open Source Projects** https://www.contributor-covenant.org/

89. **stumpsyn/policies**
GitHub
https://github.com/stumpsyn/policies

90. **A successful Git branching model**
nvie.com
http://nvie.com/posts/a-successful-git-branching-model/

91. **Kanban - A brief introduction**
Atlassian
Atlassian https://www.atlassian.com/agile/kanban

92. **GitHub features: Integrated project management tools**
GitHub
https://github.com/features/project-management

93. **Free Kanban Boards - Issue Tracker | GitKraken Boards**
GitKraken.com
https://www.gitkraken.com/boards

94. **What is Scrum?**
Scrum.org
https://www.scrum.org/resources/what-is-scrum

95. **ZenHub - Agile Project Management for GitHub** https://www.zenhub.com/

96. **Jira | Issue & Project Tracking Software**
    Atlassian
    Atlassian https://www.atlassian.com/software/jira

97. **Manifesto for Agile Software Development** https://agilemanifesto.org/

98. **c4urself/bump2version**
    Christian Verkerk
    (2021-02-13) https://github.com/c4urself/bump2version

99. **Danger - Stop Saying “You Forgot To…” in Code Review**
    Orta Therox
    http://danger.systems/ruby/index.html