Module I

Module 1: Introduction and foundations

We're reading this Module and Module 0 in tandem. In "Hello, π..." we are learning the vocabulary and grammar of a programming language, giving us the ability to express scientific ideas as code. This is sufficient for many projects, but it leaves many aspects of the craft of scientific computing — aspects which are independent of whether we're using Julia, or C++, or Python, or... — to the side. How do we organize a project so that it doesn't collapse under its own weight as it grows? How can we actually keep a meticulous record of our work, ensuring its reproducibility, when we are simultaneously running simulations and writing more code to both fix bugs and explore new corners of our models? How do we design our computational experiments so they are both scientifically sound but also computationally feasible? And what tools are available to make *collaborating* with other scientists on our work easy?

Figure 5.2: Robert Boyle (left) and Christiaan Huygens (right), the two natural philosophers involved in the earliest dispute (that I could find) in which the reproducibility of an experiment was questioned and resolved in a basically modern scientific manner [6].

This module covers the essential tools and practices in modern computational research. We'll learn about version control of our programs, scripting languages for automating analyses and simulations, the principles of reproducible research, and fundamental concepts in algorithmic complexity. This module will ensure you can manage coding projects effectively, collaborate efficiently, and think critically about the computational resources that would be needed for different tasks.

Chapter 6

Version control with Git

Version control is a systematic way of managing multiple versions of programs, of documents, of databases, and so on. Using version control makes working collaboratively with others much easier. It is also *crucial* for modern scientific reproducibility. Git is an extremely powerful system for distributed version control, and it has been overwhelmingly adopted. There are numerous guides and tutorials that will help get you up and running; my goal in this chapter is twofold. First, I want to introduce the most common, practical subset of git commands that you'll want to use from day one. Second, I want to talk about how git actually works – it does not need to be a black box, and by having a solid mental model of what happens when you use different git commands you'll avoid some of the common pitfalls that sometimes trip beginners up.

6.1 Git in practice

From a practical point of view, there are two important sets of things to learn about git. The first set is just the essential commands: How do you actually create a new "version" of your whatever your project is? How do you go back and forth between versions or compare differences between them? How do you synchronize your changes with collaborators? In the first subsection below I'll quickly cover the basics[52].

The second set is a byproduct of (a) git's "branching" model of projects and (b) the fact that git is a *distributed* version control system. "Distributed" here means that no copy of the project is more or less important than any other, except by convention. This has several nice features, but it also means there are many different patterns – often called "workflows" – for interacting with git and using it for version control. Especially when collaborating – but even when just working on your own code base – it is helpful to choose a consistent pattern of using git. While there are many different possible workflows, I think there are two that are most useful for solo projects or those involving only a small number of collaborators at a given time. I'll describe those two after covering the core commands.

6.1.1 The core commands

Nobody likes to be told to read the documentation, but the first superpower at your disposal is

[52]For alternate perspectives, consider some of the many other resources out there.

```
$ git help [command]
```

Here "`[command]`" is, not surprisingly, any git command. The output will remind you what the command does, what its options are, and other helpful pieces of information.

Getting started: init and clone

A repository ("repo") is git's fundamental unit, and each project (a paper, a codebase, a website, etc) will live in its own repo. That repo will contain all of the files associated with that project, each file's complete version history, and can even contain various parallel or alternate versions of files. You can create a new repository by moving to some directory and typing

```
$ git init
```

This initializes a repo by creating and populating a hidden `.git/` subdirectory. You can also create a new copy of an existing repo like so:

```
$ git clone [repository]
```

Here `[repository]` is either a URL pointing at a remote repo, or a path to a local repo already on your computer.

Creating versions of your project: status, diff, add, commit

Git has a "stage-and-commit" model for updating repositories. You can think of commits as the actual versions of the project you'll be able to go back and forth between, and the "staging area" as a rough draft space for the version you are about to commit. This staging area is different from the actual current state of your project, which might have many changes (new or modified files and deleted files) that are not currently staged. One reason for having this distinction is that some workflows favor having small commits that each address some specific update to the project, and using git's stage-and-commit model let's you take the many changes you may have made during a coding session and record them as a specific sequence of project versions.

In practice, you can see the current status of your project by running

```
$ git status
```

This will give an overview of your project relative to the last version that was committed: what files are in the staging area ready to be committed, and what files have been changed (or what new files have been added) but are not in the staging area. If you want more granular detail here – for instance, what lines of various files have actually been changed – you can use the `git diff` command.

To actually move a file to the staging area you run

```
$ git add [path/to/filename]
```

Turning changes in the staging area into a new version of the project is done like so:

```
$ git commit
```

By default, running `git commit` will open up a text editor asking you to specify a "commit message" – this should be a short description of the changes associated with the new version of your project.

There are *many* options for all of these git commands – and other commands you can use – that make them easier to work with. For instance, if you want to add all new and changed files to the staging area you can use `git add .` On the other hand, what if you add a file to the staging area by mistake[53]? You can "unstage" it using the `git restore` command:

```
$ git restore --staged [filename]
```

This moves the file from the staging area back into the "changed but not staged" category, i.e., without discarding your actual changes.

If you want to skip the "let's open up a text editor for the commit message" business you can use a `-m` flag on `git commit`. And if you want to skip `git add` step *for files that git is already tracking*, you can use a `-a` flag on `git commit`. This flag automatically stages all modified or deleted tracked files before committing, but it will not stage new, untracked files. So, for instance, after I finish writing this section I'll probably go to the shell and do something like:

```
$ git commit -am "added core git commands to git.tex"
```

Navigating history: log, switch, and restore

After you have committed several versions of your project to the repo, you can use the `git log` command to view the history of your project. By default it will list commits in reverse chronological order, with four pieces of information for each commit: (1) a SHA-1 checksum that serves as the commit's identifier, (2) the author of the commit, (3) the date of the commit, and (4) the commit message. While there are many options to make the log easier to parse, in practice it is typically easier to view the log through a web interface – thus, the basic command line version is sufficient most of the time.

Also, unless you're very good at memorizing hashes, write useful messages.

Having the complete history of your project would be of only limited use if you couldn't actually access past versions of your project. Here we'll use the `git restore` and `git switch` commands. To discard the changes to a specific file in your working directory (i.e., restoring it to its last committed state), you can do this:

[53] Something extremely easy to do if you go around typing `git add .` all the time!

Figure 6.1: XKCD with an astute observation about commit messages

```
$ git restore [path/to/file]
```

This is a safe and quite common way to undo unwanted edits. If you want to more specifically restore a file to the state it was in at an older commit, all you have to do is specify the source[54]:

```
$ git restore --source=[commitID] [path/to/file]
```

If you want to restore not a single file but your *entire project* to the state it was in at a past commit, you use the `git switch` command with a special flag:

```
$ git switch --detach [commitID]
```

The `--detach` flag indicates that you are entering the special "detached HEAD" state.

The detached HEAD state

When you use `git switch --detach` with a specific commit hash, you are asking Git to show you your past project exactly as it was at that time; this is very useful, but it places your repository in a special state called a *detached HEAD* (`HEAD` being special reference for git that usually points at your current position at the end of a branch). The crucial thing about this special state is that it *does not* belong to any branch – it's a floating reference that can and will be deleted by git's cleanup processes when you switch back to a real branch.
In practical terms, you need to know the following:

1. **If you just want to look around,** or run code from this point in time: you are save to do so: look at files, compile code, run tests, go wild. When you're done, go back to any branch (e.g., `git switch main`).

[54] Replacing `[commitID]` with the SHA hash of the commit you are interested in.

2. **If you want to build upon this old code:** you *must* create a new branch to save your work – creating a new branch from your current "detached HEAD" state gives this point in your project's history a name and a future. You can do this, e.g., by `git switch -c newExperimentalFeature`.

A note on the checkout command

If you look at git tutorials, you'll find a `git checkout` command that gets used for many different contexts: creating branches, restoring files, moving between commits, and so on. This one commands many different functions were a common source of confusion, and in 2019 git introduced `git switch` and `git restore` to provide more tailored commands for those actions. While `git checkout` still works (and is sometimes necessary for advanced workflows), we'll use `switch` and `restore`: they make the user's intent clearer and are safer to use.

Synchronizing clones: remote, push, pull

Using git entirely locally is itself very useful, but you will almost invariably want to synchronize your work with a remote – these are just versions of your (or someone else's) repository that live somewhere on the internet (e.g., on GitHub, or GitLab, or...). If you started a repo with `git init` you can connect it to a remote repository using the `git remote [various options]` command; if instead you started by cloning a remote repo things will be configured to synchronize easily with that remote by default. If you want you can associate arbitrarily many different remotes with your repository – this just involves many uses of the `git remote` command, and you can use `git remote -v` to see what your remotes are and how they are configured (read only, read-write).

Most of the time, you'll probably keep things simple and only have a single remote associated with your project. In this case, once you've set the remote up synchronizing with it is quite straightforward. To move your local commits to the remote you `git push`, and to take any updates that the remote has and combine it with what you have locally you `git pull`. If you want to grab all of the data the remote has but you do not want to immediately try to combine it with what you have (perhaps you're worried about incompatible changes you and a collaborator may have made to a file, and want to check everything out first), you can instead do a `git fetch`.

Creating parallel versions: branch and merge

Git's branching model makes it easy to have multiple parallel versions of your project that can develop independently from each other. It is possible because, at the end of the day, branches are just lightweight, movable pointers to a commit – cheap to make but powerful in practice. You can create a new branch – perhaps to experiment with a new feature, or develop such a feature over a long time – without interfering with whatever is going on with the main part of your project. Creating a new branch is as simple as `git branch [nameOfYourNewBranch]`, and switching between branches is as simple as `git switch [nameOfBranchYouWant]`. You can also create a new branch and immediately switch to it in a single command:

```
$ git switch -c [nameOfYourNewBranch]
```

Each branch can independently maintain its own version history (see more on that below), and when synchronizing with a remote you have complete control – you can push and pull all of your branches, a subset of them, or just have.

Eventually, you will decide that the feature you developed on some branch should be combined with your main branch (for the purposes of this guide, I'll assume that your main branch is named `main` – the actual name makes no difference to git). The basic way of doing this is to use a "merge" command: assuming that you are currently on `main` (i.e., you've recently done a `git switch main`) and you want to combine the work you did on a `featureBranch1`, all you need to do is this:

```
$ git merge featureBranch1
```

This will attempt to replay all of the changes made on `featureBranch1` on top of `main`. If there is a conflict that cannot be automatically resolved – perhaps incompatible changes were made to the same file on these two branches – the merge will stop, and you will have options for how to resolve these conflicts. I strongly recommend not beginning a merge if you have any uncommitted changes in your project.

Pull is a combination of other commands

Now that we've met both `fetch` and `merge`, I can tell you that `git pull` is essentially just a combination of these two other commands: `fetch` to get data from the remote and `merge` to combine the remote branch with your local one[a]. This is why conflicts sometimes happen during a `pull`.

[a] The `pull` command can also be configured to use a different pattern: a `fetch` followed by a `rebase`. Rebase and merge are alternate strategies for integrating changes from one branch into another. Using `rebase` – and especially its interactive version – is incredibly powerful, but it also re-writes the history of your git repo and can be dangerous. We'll leave this more advanced topic aside for now.

6.1.2 Configuring git

In addition to the basic set of git commands, there are important configurational options when it comes to setting up and using git that we should know about right out of the gate.

The first is your global `.gitconfig` file, which lives in your home directory (not your project directory), and can be created by hand or using the `git config` command. This needs to be used to set the name and email that will be associated with your commits, but can also be used to set lots of options – alias for commands you want git to use, the default editor to use when writing commit messages when not using the `-m` flag, and so on.

The second is the `.gitignore` file, which is a per-project file located in the root of your repo which instructs git that, by default, certain files should not be added to your repo. For

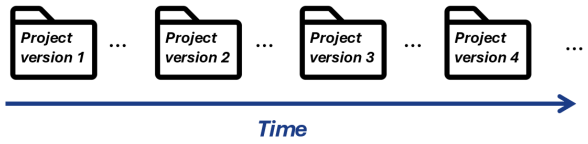

Figure 6.2: Schematic view of a project with a centralized workflow.

instance, in a repo containing a LaTeX document you might not want to version control all of the ancillary files that TeX generates as it compiles your document, so you might write a `.gitignore` file like this:

```
## ignore core auxilliary files
*.aux
*.log
*.out
# ... other files
```

If you compile your TeX document (so files like this exist) and type `git add .` you will find that files matching either the specific names or the patterns you have written in the gitignore file are *not* moved to the staging area. You can force git to add an ignored file (using `git add -f`); the `.gitignore` file just controls the default behavior. This is extremely useful for keeping a clean repo, only version controlling the files for which version control is appropriate.

6.1.3 Common workflows

Patterns of interacting with git can get quite involved, especially as the size of a project and the number of contributors to it grow. For the kind of smaller-scale projects we will be working with, the following two simplest patterns will serve us well.

Centralized workflow

The first is usually called the *centralized workflow*, in which we ignore git's branching model altogether and have all of our project's history linearly recorded on the repo's main (i.e., only) branch. Schematically, our project's history will look like that of Fig. 6.2.

In this schematic time progresses from left to right, and I've deliberately used dots to connect the project versions themselves rather than the arrows you might have expected. The reason for this choice will be clear when we talk about how git stores a repository in Section 6.2, and we realize that arrows connecting the different versions should be drawn opposite to the flow of time.

This workflow works well for small computational projects – perhaps small analysis scripts, or collections of related Mathematica / python notebooks – and also especially for working on papers (which should definitely be version-controlled with a method better than saving files with names like `manuscriptDraft_v10_final_revision_v2.tex`).

Centralized workflow

For concise reference: the basic pattern of operation with this workflow is

1. Initialize the repo: `git init` or `git clone`
2. Make and commit changes: `git status`, `git add`, and `git commit`
3. Synchronize with the remote
    - Get any changes (if you are collaborating): `git pull`
    - Handle any conflicts, if necessary
    - Push changes: `git push`
4. Repeat steps 2 and 3 until done.

This simple pattern uses only a handful of git commands, but is already very useful. It can also be meaningfully thought of as a building block for more complicated ways of working with a repo. The next example demonstrates this, where I'll use the abbreviation CWF to refer to steps 2–4 above.

Feature branch workflow

As projects get larger, the main branch of your project using CWF can start to get cluttered and chaotic. Perhaps you're building a particle-based simulation framework, and you are simultaneously adding new forces and equations of motion and boundary conditions, all while occasionally finding and fixing a bug or two. The commits for these additions are all interwoven in your project's history – does a particular new feature rely on a bug fix earlier in the commit history, or not? Are the changes you needed to make to a file containing a common `simulationFramework` class when working on two different features compatible, incompatible, or completely independent from each other?

A nice pattern for encapsulating the work on separate parts of your project is called a "feature branch workflow. It is based around having a primary `main` branch for your project, and instead of committing directly to it you are encouraged to create a new branch when you want to start working on a new feature. These feature branches get merged with the main branch when they are ready, and then they don't need to be touched again (again, since branches are just lightweight pointers, there's no real cost to having a bunch of unused ("stale") branches around). It looks schematically like Fig. 6.3.

I use this pattern for my open-source scientific code packages, where I want the main branch to always (hopefully!) have working code that others can use. This is one of the main benefits of a feature branch workflow: you can be developing and testing multiple items independently, and the main branch can stay clean and functional. Yes, I still sometimes commit to main when implementing a quick bugfix, but the basic pattern of "branch for a new feature, merge when it's done" is very convenient.

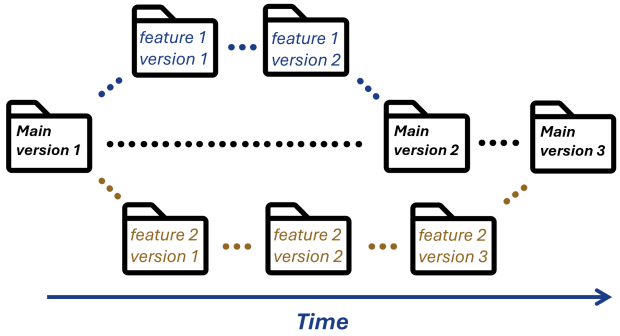

Figure 6.3: Schematic view of a project with a feature branch workflow

Feature branch workflow

The corresponding basic pattern of operation is something like:

1. Start from the main branch (perhaps after some period CWF development): `git switch main`.
2. Create a new branch for a new feature: `git switch -c nameOfNewBranch`
3. Work on that feature branch using a CWF
4. Bring your changes back to the main branch: `git merge`

One thing to note is that this pattern tends to favor relatively small and shorter-lived branches, since merge conflicts get more common and can be harder to resolve the longer a branch has diverged from main. Finally, I'll note that a feature branch workflow can itself be a building block for more complex workflows. The complexity of your workflow should probably scale[55] with the size of your project and the number of collaborators.

6.2 How git stores a repository

It helps to know about how git actually storing your project and its version history. This lets you know what all of the commands in the previous section are actually doing, and it also helps you correctly reason about what will happen when you make a commit, or when you want to merge two branches. So: git is a directed acyclic graph – the nodes of this graph will be a small number of different types of "git objects", almost all of which can then point to other git objects (forming the directed edges of the graph). We'll see how this turns into a system for version control by meeting the important git objects.

6.2.1 Blobs and trees

Blobs

[55] Logarithmically?

The most basic git object is the *blob*, which represents the *contents* of a file.We'll represent them like in Fig. 6.4.

Figure 6.4: A blob object. Mr. Blobby photo credit: Kerryn Parkinson/NORFANZ

What's going on here, and why have I written `d23a1` on the blob? When we `git add` a file (or when a merge changes an existing file's content), git reads that file into a memory buffer and uses a lossless data compression algorithm to figure out what the compressed version of that file is. It then prepares a file which contains a short header followed by that compressed representation of the file. Git next calculates the SHA-1 hash of the blob, and uses the 40-hexadecimal-digit representation of the hash value as the *name* of the blob file, which then gets stored in the `.git/objects/` directory (technically, git uses the first two hex digits of the hash as a subdirectory name and the remaining 38 digits as the file name). The header schematically looks something like this:

```
blob (size of compressed blob in bytes)
(binary representation of compressed blob)
```

That is, it has information that this is a "blob"-type object that will be of a certain size, followed by the compressed version of the file. I don't want to type 40 digits for the names of blobs, so I'll just use 5 letter/number combinations to represent these hash values, as in `d23a1` above[56].

Trees

You may have noticed that nowhere in the blob is there information about what the file is named, or where to find it relative to your project's root directory. Perhaps you are extremely good at memorizing SHA-1 hashes, but the rest of us would probably like to go on using file names and paths as usual. The next kind of object that git uses is a *tree* object – these are objects who's purpose is to point to other blobs and trees, and associate the usual information that we think of when we work with files in a file system. In that sense they are like directories and subdirectories. A visual representation of such an arrangement is in Fig. 6.5.

There is nothing mystical about all of these arrows pointing from trees to other objects; schematically a tree object is represented in a file as something like:

```
tree (size of tree in bytes)
(mode) (blob file name) (blob objectID)
(mode) (blob file name) (blob objectID)
(mode) (tree path name) (tree objectID)
...
(mode) (blob file name) (blob objectID)
```

[56]SHA-1 is a useful hashing algorithm, but it is not cryptographically secure against "collision attacks," where an attacker could craft two files that result in the same hash. For the purposes of version control and protecting against accidental corruption, it remains perfectly suitable.

That is: each tree contains a header (this is a "tree"-type object that will be of a certain size) followed by a list of things the tree points to. Each item in that list has a "mode" – is it a normal file, an executable file, a symlink, a type of directory, a git submodule – a file/path name, and finally an object ID. It's in the sense that the header of the object contains the type and ID of other objects that a node in git's graph "points" to another node.

Naturally, the SHA-1 hash of the tree object gets used as the tree's object ID.

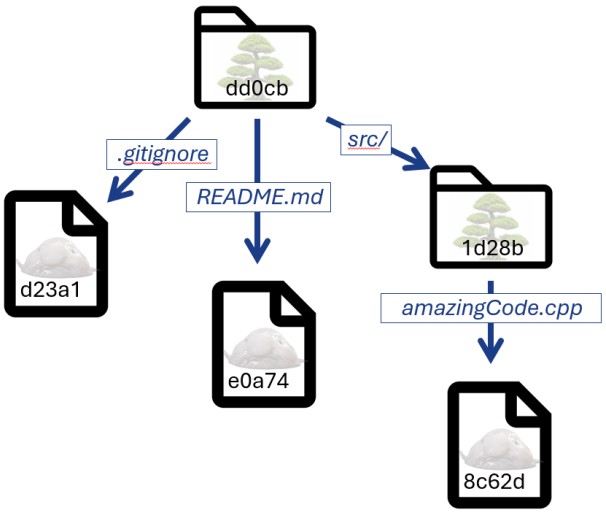

Figure 6.5: A tree object pointing at blobs and trees

6.2.2 Commits

Thinking about the above, we see that we could create different versions of a project by being able to point at different tree objects – in the above image, if I could remember the `dd0cb...` hash I would be able to find the file corresponding to that tree, and from there get all of the sub-trees and blobs that contain information about the state of the project at that time.

Again, unless you are extremely good at memorizing SHA-1 hashes, you probably want a new type of object for this purpose; that is what a *commit* object is for. The format of a commit object is schematically

```
commit (size of commit in bytes)
tree (tree's object ID)
parent (parent commit's ID)
...
(commit information)
```

Yet again we have a header saying that this is a "commit"-type object of a certain size. Here that header is followed by the relevant information about the commit. This includes the tree that itself points to the blobs and trees that make up the state of the project, along with information about any "parent" commits. Typically a commit will have one parent commit, but (a) the first commit of a repository will have zero parents and (b) when merging branches a commit can have multiple parents. Finally, there is a bunch of additional information about the commit: the author, commit date, the commit message, and so on. Because all of this information is used to generate the commit's hash changing anything – even a single character in the commit message! – will result in a completely new commit with a different hash. By now, you will not be surprised to learn that a commit objects' ID is just the SHA-1 hash of the commit object.

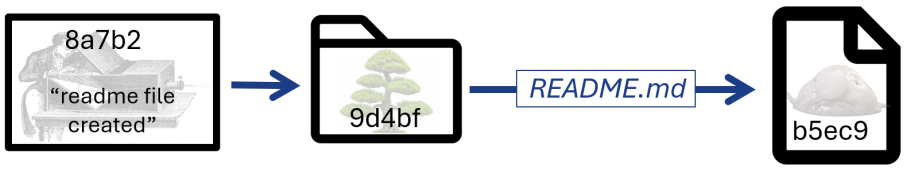

Figure 6.6: The state of the repo after our first commit.

A sample repo over time

Bringing these three basic git object types together, let's see what our repo looks like over the course of a few simple commits. Below I'll go back and forth between simple commands at the shell and a visual representation of the repo. It is implied that whenever the shell command is `nvim [some file]` I am creating or editing that file and saving it. We'll start out simply: in a completely empty directory let's initialize a git repo, edit a single file, then add and commit it.

```
$ git init
$ nvim README.md
$ git add .
$ git commit -m "readme file created"
```

After this, Fig. 6.6 shows what our repository looks like.

Pretty simple: a single commit represented as a snapshot[57] which points at a tree, which points at a blob. Let's add a little bit of complexity by adding a new file in a new subdirectory of our project:

```
$ mkdir src
$ nvim src/amazingCode.cpp
$ git add .
$ git commit -m "code added"
```

Now our repository looks like in Fig. 6.7.

There are a few things to notice. First, as promised, the new commit points both to the parent commit and to a tree. Second, git is happy to re-use any existing data it can: here, the `README.md` file didn't change, so the same blob object is pointed to. On the other hand, the *tree* at the root of our project did change: it contains the file it already had *and* a new sub-tree. Thus, the new commit cannot reuse the original root tree.

To advance one step further, what if we make a new commit that (a) adds a file and (b) edits an existing file? Something like:

[57] Depicted with a drawing of a camera obscura... It's no "blobfish for a blob", but it gets the job done.

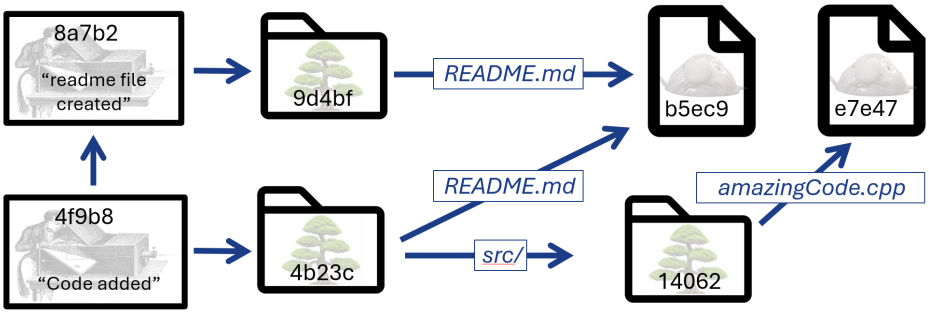

Figure 6.7: The state of the repo after a subfolder and new file are added.

```
$ nvim .gitigore
$ nvim README.md
$ git add .
$ git commit -m "gitignore added and readme edited"
```

Based on what we know, we expect the following. The `src/` directory and its contents haven't changed, so the new commit should point to a tree that points to the *same* `src/` tree as in the last illustration. The root tree that the commit points to should be different, because it needs to be a tree that points at two blobs and one tree (unlike the "one blob and one tree" root tree of the previous commit). Finally, we should see an entirely new blob appear, corresponding to the contents of the edited `README.md` file. Indeed, this is what we have (see Fig. 6.8).

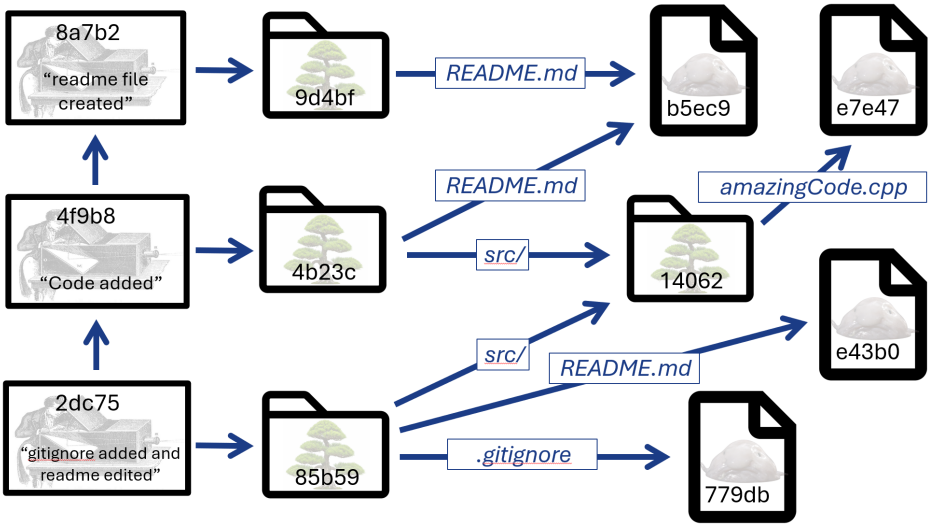

Figure 6.8: The state of the repo after a new file is added and an old file is edited.

It is worth emphasizing again that there are now two different blobs corresponding to the two different versions of the `README.md` file in the repository. And, since both are reachable in the graph from the `2dc75` commit, you have access to both of them. Exactly as you would hope for a version control system.

6.2.3 References

There is one more class of git object to meet as we finish things up, and these are git references – HEADS, tags, and remotes. References are pointers, acting like sticky notes that can tell you where you currently are in your project's history, noting an interesting commit (or, in fact, any interesting node in the graph), or noting an objectID on different clones of your project.

First: where are you currently in your project's history, and what commit do you want to base your next commit off of? You probably don't want to memorize the SHA-1 hash of the answer to this (something of a recurring theme in this section), so git maintains a list of HEAD references (these are files in the `.git/refs/` directory, one for each branch). Each of these files just contains the SHA-1 hash of a commit object corresponding to the current snapshot of the branch in question.

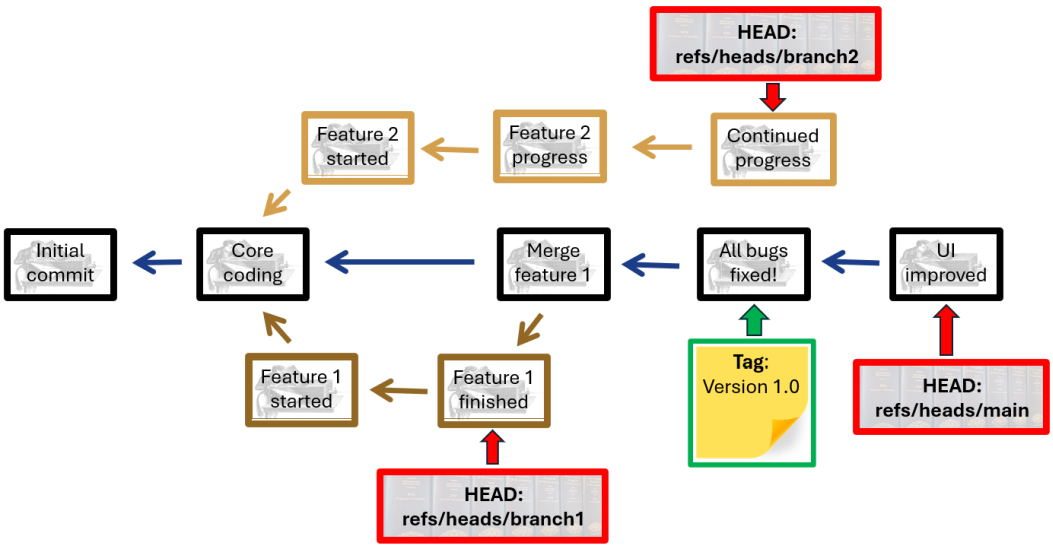

Figure 6.9: Schematic view of a project with HEADS and tags.

Second: you might want to have some extra mechanism for pointing at specific object in your project's history – perhaps the commit you want to correspond to version 1.0 of a code release, or the first submission of a paper and then the finalized revision after you get the referee reports sorted out – and git provides "tags" for this purpose. Basically, a tag is just a time-stamped message that points to a specific commit. The storage format is similar to that of a commit object; technically tags can point at anything (important blobs, or important trees), but those use-cases are probably not something you need to worry about right now. I certainly don't.

A simplified view of a remote with a few branches and a tag might look something like in Fig. 6.9.

Finally, there are remote references. These contain the object ID of the HEAD of the various branches on your remote(s) *the last time you communicated with the remote server*. You will probably not really ever need to look at these remote references (which are created in the `.git/refs/remotes/` directory when you set up a remote), but if you do and you want up-to-date information, you'll need to `git fetch` first.

Git and compression

Git stores the contents of files in a compressed representation. For for files that have changed, git will periodically do its best to represent these different versions not as totally independent blobs which are mostly the same as each other. Instead it will (schematically) try to store them as a base file and a sequence of minimal changes needed to move between different versions of it[58]. The upshot is that if your repository is mostly just text files and perhaps a few images (as it might be for some code, or when writing a paper), you absolutely do not need to worry about how much space and overhead git uses to implement the model of version control described here. On the other hand, occasional changes to large files – for instance, to videos – can cause a repo to quickly grow in size. At a minimum, every file in your project that you track with git requires both the space for the file itself and for git's compressed blob representation of it that sits in the `.git/` directory. For text files this is a trivial addition, but for already-compressed video formats this might roughly double the amount of storage space you are using.

[58] For full details, you can read about git's packfiles.

Chapter 7

Blueprints for a computational research project

Organizing successful scientific projects often requires planning on three different levels: there are practical nuts-and-bolts decisions to be made about the organization of its files, there are choices that must be made about the core tools you will use and the algorithms you will employ, and there is planning to be done in structuring the way you will use those tools to analyze data or conduct numerical experiments. Oftentimes all of these aspects organically grow over time with the project, but it is extremely helpful to think through each of them even before you write a single line of code.

The first two sections below will be extremely practical: what are the nuts and bolts of how to organize the files in your project, and how can you set up scripts that will actually run your code in a reproducible way? This material is not complicated, but it is often simply not mentioned at all. The last section will be somewhat more general: if the recurring theme of this course involves the composition of algorithms and data structures to solve problems, what are the kinds of criteria should we actually use to choose our algorithms?

7.1 Organizing your project

For many of the projects in this course – and in your own research! – you will be developing a core set of Julia code, running it with various scripts, generating data, and then producing plots or other results from that data. A scientific paper may or may not ensue. Given this common pattern, it makes sense to organize each project in a broadly similar way. By doing so, a collaborator (or a future version of yourself, after you have forgotten all of the nitty gritty details) should be able to quickly figure out what you did in your project, and why you made those choices.

7.1.1 File organization

Some of the details will, of course, be particular to your project and the tools that you use in it, but most of it will not be. A structure[59] that handles most reasonable scientific workflows often looks something like code block 7.1.

```
MyResearchProject/
├── data/
│   ├── README.md
│   └── awesomeData.h5
├── plots/
│   └── coolPlot.png
├── research/
│   ├── README.md
│   └── interestingDerivation.tex
├── scripts/
│   ├── README.md
│   └── runSimulation.jl
├── src/
│   ├── MyResearchProject.jl
│   └── (other source files, like greatFunctions.jl, etc.)
├── .gitignore
├── Manifest.toml
├── Project.toml
└── README.md
```

Code block 7.1: A sample structure for organizing a scientific project

Let's break down what's going on here. First, the directory itself has a sensible name, allowing me to identify the purpose of the directory from the command line / a file explorer. The `src/` directory is where the core logic of the project lives. For a Julia project this will contain the source files (with extension `.jl`) that define your primary modules, types, and functions. We'll see in Chapter 5 that Julia's package manager can generate this directory (and some of the files in the root directory that we'll talk about in a moment) for you.

The `data/` directory is... well, a dedicated directory for the data associated with the project. You should *always save the raw data*, but sometimes extremely large datasets associated with a project might not fit nicely into this tidy directory structure (and, for instance, if you're hosting the repository on GitHub the files might just be too big). Processed data, and data that can be directly used to generate plots, should probably live here.

[59] Just one of many, of course. For reasonable alternatives that span a range of disciplines you might see, for instance, this guide to organizing projects in computational biology, or this generic git template, or the default project structure suggested by the DrWatson Julia package.

Do not version control large data files!

Large data files should not be tracked by Git. Version control is for code and text, and not binary blobs of data. Adding such files will bloat your repository, making it slow and difficult to work with. Thus, your `.gitignore` file should include entries that ignore large data files; a `README.md` file inside the `data/` directory the correct place to document the origin and current location of your data.

While `src/` contains the reusable, library-like code, the `scripts/` directory holds the "executable" scripts that *use* that code to perform specific tasks. A script might run a simulation with a specific set of parameters, or take processed data and generate a plot for a paper. As a rule of thumb: if it's executable code that *generates* another file on your computer, you should probably think of it as a script.

The `plots/` directory is pretty self-explanatory: it's a dedicated place for figures and other visualizations that you generate. Like data, it is often reasonable to think of these as *products of your code* and hence can be safely ignored by Git. The scripts that access the data in the directories described above should be trivially able to regenerate plots whenever you want.

The `research/` directory is where I like to keep notes, relevant derivations, to-do lists, related papers, and so on. Think of it as a digital lab notebook for the project, containing the messy, exploratory work that might one day inform a more polished manuscript – I want a record of all of this work, but I don't necessarily think all of it will end up being core to the project. If the scope of the project is *extremely* clear – that is, there is no way that the project corresponds to anything other than exactly one paper – I sometimes add a 'paper/' directory with all of the LaTeX, bibliographic information, and final figures. It is rarely so clear, and I almost always just have a separate repository for the paper.

At the root of the project's directory there are some configuration files and some documentation. We'll learn more about the `Project.toml` and `Manifest.toml` in Section 5.2, and these are specific to Julia[60]. There is of course a `.gitignore` file, which tells Git which files and directories to ignore.

Finally, notice how this and, in fact, almost all of the directories have a `README.md` file. In the root of the directory this is the "front page" of your project, and should explain what the project is, how to install any dependencies, how to run the code, and so one. GitHub is set up so that a readme file will be presented with nice markdown formatting if one is in any subdirectory, though. Since having a human-readable summary is nice when navigating online, it's worth learning a little bit of GitHub flavored markdown for this task.

The key principle in this project structure is a separation of concerns. Source code is separate from the scripts that run it; scripts are separate from the data they produce and the plots that visualize it.

7.2 Planning and executing computational experiments

The `scripts/` directory is where the logic of your code meets reality. What set of parameters should I sweep over in order to solve my problem, and how large is the set of parameters that

[60] If this had been a C++ project, there might instead be a `CMakeLists.txt` file here, for instance.

I actually have the resources to sweep over? Is one of those sets larger than the other? How should you call the code you wrote in order to perform numerical experiments that are not only robust but also *reproducible*?

7.2.1 Fermi estimation for your code

At the risk of being overly explicit by doing arithmetic in front of you – what follows is going to be both extremely simple *and* extremely powerful at a practical level – let's actually connect the idea of algorithmic complexity to the practice of designing numerical experiments.

Suppose we're running a simulation of the stars in a galaxy, and we realize that to evolve the motion of $N = 10^3$ stars forward in time for the equivalent of 7 year takes about 3.3 seconds. We've also run some small tests on our code and have found that its runtime scales *quadratically* with the number of stars – that is, the time taken is proportional to N^2 – a scaling that is common for direct implementations of algorithms that involve checking all pairs of interactions. As we will formalize in Section 7.3, this scaling behavior is usually denoted using "Big-O" notation and written $O(N^2)$; for now, all we need is the observed quadratic relationship to make some sensible estimates.

So, with what we have so far – and assuming the runtime scales *linearly* with the forward simulation time – how long will it take to simulate the motion of $N = 10^6$ stars[61] for the equivalent of 100 years? Our first estimate should be approximately $(3.3\text{s}) \times (100/7) \times (10^6/10^3)^2 \approx 4.714 \times 10^7$ seconds (i.e., about a year and a half). Do you have that much computer time available (and do you need to graduate before then?)? Is a simulation covering 100 years even close to what it takes to answer your question, and do you need to find a different way of tackling your problem?

Those are the kinds of questions whose answer depends on the science, on your resources, and on what you can actually code up. But making these back-of-the-envelope estimates of how long it will take to do something given the tools you already have is both a crucial skill and is independent of those answers. In addition to using these estimates to figure out whether you can, e.g., get something done by a certain deadline or given a certain amount of resources, another place these estimates always come up is in determining parameter sweeps. Perhaps you want to study how some material behaves as you control density and temperature. Even if you know what range of density and temperature you want your simulations to span, how many simulations should you actually launch? Can you estimate how long each simulation will take to finish based on the asymptotic scaling of the algorithms and what you know about how long you need each simulation to run for? Combining those estimates, and thinking about whether you will have each simulation run independently of the others (as, for, instance, if you were just planning on simulating the parameters on a regularly spaced grid between the endpoints you already decided on) or not (as, for instance, if you want to use information from one simulation to help decide what point in parameter space would give you the most information), is a bread-and-butter part of computational research.

[61] Which could, perhaps, be a reasonable estimate for some dwarf galaxies.

7.2.2 Designing scripts for reproducibility

We often want to write functions that we will call many times as we vary key parameters of some physical model. There is a subtle danger in the fact that Julia is a language in which it is easy to write both robust, performant code and disposable interactive scripts. Namely: it is so easy to iterate and quickly generate results, that it arguably requires *more* care to generate reproducible results.

This can be seen by contrasting with a compiled language like C++. In such languages there is a clear separation between being in the phase of working on code and being in a phase of *using* that code: the clear separation is generated by needing to compile the source code into an executable. Thus, a natural workflow involves working on the code itself until it compiles and functions correctly, making sure the program can accept command-line arguments for the parameters of interest, and then writing a script in some other language – bash or python, perhaps – that calls that code repeatedly across the array of parameters. This leaves behind multiple artifacts that can be version controlled and referred to later. Do you want to reproduce exactly the results of some paper? Go into the git repo, checkout the commit where you checked in the calling script (which will ideally also restore the codebase to its state when that script was run!), and then run that script.

The fact that Julia's workflow is so fluid means that we – the programmers! – must provide the discipline that a compile-step enforces in other languages. Let's walk through a progression of different patterns we might use to work with Julia in the context of scientific computations. Each pattern might be reasonable in different contexts, but we'll see what kinds of problems and pitfalls each might entail. All of this will use, as an example, the Julia file in code block 7.2, which defines a simple `estimatePi` function that takes two parameters.

```
# pi_estimation.jl
# (Our Monte Carlo estimate_pi function from before)
using Random
using Statistics
generate_points(n,L) = [(rand(Float64,2) .*L .-L/2) for i in 1:n]
in_unit_circle(point) = sum(point .* point) < 1.
function unit_circle_proportion(points)
    return count(in_unit_circle,points)/length(points)
end
function estimate_pi(n,trials)
    data = [ 4 * unit_circle_proportion(generate_points(n,2))
             for _ in 1:trials]
    return (mean(data),var(data))
end
```

Code block 7.2: The beginning of a script containing functions for a Monte Carlo estimation of π (see Section 4.3).

Workflow 1: The interactive session

The REPL is an essential, powerful tool for exploration, prototyping, and debugging. The challenge is in capturing the final, successful version of the process we followed in permanent form.

Consider the following sequence of events. The very first thing we might find ourselves doing is in the spirit of the `Revise`-based workflow[62] we discussed in Chapter 1. Earlier in our Julia session we had done[63]

```
julia> includet("pi_estimation.jl")
```

As soon as we were happy with the state of the functions we could start generating data, for instance

```
julia> first_estimates=estimate_pi(10000,15);
```

Do that a few times, and we're already able to start generating a plot with data to use!

All of which is to say: in a language like Julia, the transition from writing code to testing it to generating data is much blurrier. The instant you think you have some working code you might start calling the relevant functions from the REPL, saving data, and so on. It feels amazing, but how are you going to instruct someone else to reproduce that data? Do you remember exactly what you typed into the REPL? Are you sure that you didn't revise a function, run something in the REPL, revise a function again, and run the same thing in the REPL, and then undo that revision before committing to the git repo?

To be explicit, a REPL-driven workflow, for all of its power, presents two major challenges to scientific reproducibility: *provenance* (what is the exact state of the code that produced a result) and *history* (what were the exact sequence of commands and parameters that were run).

Workflow 2: The self-contained script

A crucial first step that solves these problems is to create a dedicated script whose sole purpose is to run our experiment. After we create it, we add it (and the rest of the state of our codebase) as a new commit to our repository. This gives us a permanent, version-controlled artifact that records exactly the computation that was performed. It might look something like this:

```
#run_single_pi_estimation.jl
include("pi_estimation.jl")
n_points = 10000
n_trials = 10

mean_pi, variance_pi = estimate_pi(n_points,n_trials)
println("Parameters: N=$n_points, Trials=$n_trials")
println("Pi estimate: $mean_pi (variance: $variance_pi)")
```

[62] Even before this, I suppose, we might define *all of our functions* directly in the REPL. Let's not do that.

[63] Or, even better, we had written a module for our whole π estimation project, and we were `using` that module.

Here we've written a script that prints the output to the screen – in a real script you would, of course, save the data to a file. To use this script we just call it from the command line:

```
$ julia run_single_pi_estimation.jl
```

This is already an improvement over the REPL-based approach, as it leaves us with a permanent record of what we did.

On the other hand, it is true that every time we want to run our experiment with different parameters we have to go in and edit this file. We can avoid this – at the cost of potentially losing out on some of the provenance of our results – passing parameters from the command line. Julia makes this easy: when you call Julia with command line arguments those arguments populate a global `ARGS` constant, and one can then parse this constant to extract information[64].

```
#run_and_save_pi_estimation.jl
include("pi_estimation.jl")
n_points = 10000
n_trials = 10
seed = 1234 # a poor default

#fragile parsing of command line arguments
if length(ARGS) >=1
    seed = parse(Int,ARGS[1])
end
Random.seed!(seed)

mean_pi, variance_pi = estimate_pi(n_points,n_trials)
filename = "./pi_N$(n_points)_T$(n_trials)_seed$(seed).txt"
open(filename, "w") do f
    write(f, "$mean_pi, $variance_pi")
end
```

Code block 7.3: A script with very simple command-line-argument parsing.

Let's use this to solve *another* problem with the above script, which is that it is not actually reproducible at all! It makes use of Julia's RNG, and if you have your friend clone your git repo and run the script from the precise commit in question, they will *not* get the same results as you[65]. We'll learn more about this in **??**, but for now let's tentative content ourselves with thinking that by setting a "seed" – an initial value of some sort – we'll ensure a reproducible, deterministic sequence of random numbers. Our revised script may now look like code block 7.3.

[64] If you find yourself parsing command line arguments frequently, consider using dedicated packages (e.g., `ArgParse.jl`) to handle many of the details for you.

[65] Barring, of course, a coincidence of astronomical unlikelihood.

This embodies a few best ("reasonable") practices. It explicitly seeds the random number generator, ensuring that anyone that runs it with the same seed will get the same numerical result. It can be run with default parameters:

```
$ julia run_and_save_pi_estimation.jl
```

Those parameters can be overridden from the command line:

```
$ julia run_and_save_pi_estimation.jl 9823475
```

It then saves the output to a systematically named file, so that from the filename itself we can reconstruct what parameters were used to generate it. That is, even if we extended the command line argument parsing to include passing in the other parameters of our experiment – `nPoints` and `nTrials` – and then ran everything from the command line, we would still have a record of what experiments were actually run.

Workflow 3: The parallel-execution strategy

What we have above is already quite powerful. It would also be quite tedious – typing in variations of command line arguments over and over is both tiresome and error-prone. We could of course just hard-code the complete set of parameters we want to sweep over, but we have still limited ourselves to a fundamentally *serial* execution of our experiments. For our example of estimating *pi*, each experiment is independent of the others, but we still have to wait for each one to finish before the next begins. On modern, multi-core computers this leaves a huge amount of computational resources we could be exploiting on the table.

Fortunately, Julia has *fantastic*, built-in support for multi-threading. A common pattern would be like in code block 7.4, where we use this built-in support to perform our parameter sweep for us. The only thing we need to tell it from the command line is how many threads to use, for instance like this:

```
$ julia --threads=4 run_pi_parameter_sweep.jl
```

This is incredibly powerful.

By adding the `@threads` macro to our `for` loop, Julia automatically divides the work among the available threads[66]. This is a form of *data parallelism*, which is safe in this example because each iteration of our loop is completely independent – each run works on its own parameters and saves results to a unique file.

[66] If, on the other hand, we set the `threads` option to 1 (or if we just don't specify it at all) Julia understands that the number of threads available is 1 and then happily returns to running the loop serially for us.

Parallel programming

The simplicity of `@threads` is deceptive, and it should *only* be used in loops where the iterations are truly independent. If they are not, you will encounter a *race condition*: the output of your program will depend on the unpredictable order in which multiple threads or processes access and modify shared data. This can lead to silently corrupted data, incorrect results, and non-reproducible errors that are extremely difficult to track down. We're using `@threads` in a carefully controlled, safe context here. True parallel programming requires much more advanced patterns.

Another caveat to using `@threads` is its scheduling behavior. The *scheduler* determines how the loop's iterations are assigned to available threads, and the default[67] breaks the work into small chunks and uses a shared queue. This is a good, all-around choice that keeps threads busy even if the work per iteration is unbalanced. Note, though, that the order in which these small chunks are processed is not guaranteed. Two main alternatives exist. If you want slightly stronger guarantees about which thread processes which iteration you can use the `:static` scheduler (`@threads :static for...`), which evenly divides the iteration space and gives each thread a large continuous chunk of work to do. For workloads with high variability from one iteration to the next, the `:greedy` scheduler can be good: as soon as a thread is free it just grabs whatever the next available iteration happens to be.

Relative paths for files

Another best practice embodied in code block 7.4 is the use of *relative paths* when saving files[a] – you can see this in the dot slash at the start of `./data/...`. Avoid hard-coding directory paths if you can, so that your script will work regardless of what computer it is run on. Note, by the way, that you can use the `mkpath()` function to safely create directories so that you don't have to worry about whether they already exist

[a] While building path strings works well, it doesn't always play nicely across operating systems. Julia has a built-in `joinpath()` function we can use, like so:

```
filepath=joinpath(".","data","pi_N$(nPoints)_T$(nTrials)_seed$(seed).txt")
```

This automatically uses the correct path separator ("/" or "\") for the operating system.

Workflow 4: The managed project

Honestly, the pattern demonstrated in code block 7.4 is what I've used for the majority of my own computational projects over the years – not necessarily in Julia, but one can easily build up similar patterns with, e.g., bash scripts for running an executable program that might or might not take command line arguments. Often, as you write scripts like this over an over, you can find your self building progressively more intricate functions of convenience: perhaps a function that will nicely generate filenames for saving data for any number of parameters you

[67] Which currently corresponds to typing out `@threads :dynamic for...`

```julia
#run_pi_parameter_sweep.jl
using Base.Threads
include("pi_estimation.jl")

# The grid of parameters we want to explore
parameter_grid = [(1000, 100), (5000, 100), (10000, 50),
             (20000, 50), (50000, 20), (100000, 20)]

@threads for (n_points, n_trials) in parameter_grid
    # A simple way to get a unique seed per run
    seed = n_points + n_trials
    Random.seed!(seed)
    mean_pi, variance_pi = estimate_pi(n_points, n_trials)

    #Save results
    filename = "./data/pi_N$(n_points)_T$(n_trials)_seed$(seed).txt"
    open(filename, "w") do f
        write(f, "$mean_pi, $variance_pi")
    end
end
```

Code block 7.4: A script that loops over a grid of parameters, using the number of threads available to potentially run iterations of the loop in parallel.

want to vary, or routines that perform a simulation only if the output of that simulation doesn't already exist[68], and so on.

There is also a kind of fragility to these kinds of scripts. The way we parse the command line arguments is extremely fragile, requiring us to invoke all commands in exactly the right order. We have to manually create the `./data/` directory *before* we run our script, otherwise it will fail (perhaps in one of the worst ways: running all of the expensive computations but then silently not saving any of the data because the path to the target file doesn't exist). All of these specific problems are, in fact, easily solvable in Julia. But the broader point is that, in fact, this entire broader class of problem is not only solvable but *already solved* – we could solve them, but this is just creeping up to the edge of re-inventing the wheel.

The specific solution you want to use here is often language (and sometimes domain) specific, so I don't want to dwell overly long on the details of using what are known as *scientific workflow systems*. It's more important simply to know that they exist; when your project workflow starts becoming long enough, or you find yourself starting to write a lot of boilerplate code in the scripts that are running your main code for you, that's a sign you should investigate more. In the Julia ecosystem, a commonly recommended tool is the "scientific project assistant," `DrWatson.jl` package. It is designed to solve exactly these problems of fragile, boilerplate code: it provides standardized functions for accessing data paths, running files from parameters, and

[68] Very handy, e.g., for dealing with the occasional dropped job when running thousands of simulations on a cluster!

safely running experiments, all while integrating with the reproducible project structure we discussed earlier.

7.3 Algorithms and the logical architecture of a project

"The `src/` directory is where the core logic of the project lives." But even before we write a single line of code in `MyResearchProject.jl`, we need to think about one of the most consequential decisions in our project's lifecycle: what set of algorithms we will use to accomplish our goals. It is often the case that there are multiple ways to *numerically* implement the solution to those goals – perhaps a "brute force" method along with an increasingly slick set of "clever" methods. Where should we start, and how can we be quantitative about making this decision? How should we weigh the pros and cons of using "better" algorithms with things like the amount of time it will take us to code them, or the number of bugs per line of code we expect to have lurking in our project?

7.3.1 Algorithmic complexity

This is just an introduction!

The field of algorithmic analysis is a deep, fascinating discipline in its own right! It would be impossible to provide a comprehensive treatment of the subject in just the few pages I've written here, and that is not my goal. Instead, I will introduce the essential concepts and the vocabulary that will let us reason about our choices and make informed decisions as practicing scientists.

Time (how long it will take to run an algorithm given some data) and *space* (how much memory the computer will need) are two of the most fundamental computational resources we need to think about. Occasionally it is useful to think of these quantities in absolute terms – how many days will it take once I start this code for it to finish? More often, though, we don't really care about those questions, as they are too dependent on all of the details[69]. It is often better to ask how a particular algorithm *scales* with the size of the problem. For instance, if you are simulating a galaxy with N stars and then decide you want to simulate twice as many, how much harder does the problem get? Will the solution to a system of $2N$ stars take twice as long? Four times as long? Exponentially longer? Similarly, we can ask about *space complexity*: will the simulation require twice as much memory, or will the memory requirements grow more steeply? In the following I'll focus on the time complexity, but the memory footprint of a chosen approach can be equally important.

In order to explain the answer to such questions, it is common to use asymptotic notation. If we have some function which captures, say, the running time of an algorithm as a function of the "problem size" (for instance, the number of stars we are simulating, or the number of

[69] What CPU does your computer have? What other processes might be running at the same time, sharing both CPU cores and RAM?

elements of a list we want to sort, or...), $f(N)$, we might write:

$$f(N) = O(g(N)).$$

This means that *in the limit* where $N \to \infty$, the ratio $f(N)/g(N)$ is finite[70]. Most commonly the function g will things like a constant ($O(1)$), a log ($O(\log N)$), a polynomial ($O(N^p)$), or an exponential ($O(2^N)$).

This "big-O" notation is all about the asymptotic behavior of an algorithm, *and* there is an important hidden prefactor (the precise ratio between f and g): it could be the case that for a particular problem size an $O(1)$ algorithm is actually slower than an $O(N^2)$ algorithm. It is often also important to distinguish between the asymptotic behavior of an algorithm given "typical" or "average-case" data to work on, vs the asymptotic behavior of an algorithm in the worst possible case. But even with all of these caveats, understanding the asymptotic behavior of algorithms is a crucial first step. If you are working with an $O(N)$ solution to your problem, you can be confident that if you later realize you need to work with a problem which is an order of magnitude larger you'll be okay. On the other hand, if your solution to the problem is $O(2^N)$ and you have the same realization...well, I hope you're prepared to be a student for an extremely long time!

As a final comment: time and space are just two of many resources that may be important to think about. Modern computers – not only supercomputers but also consumer laptops – have multiple processing cores and specialized GPUs that often offer thousands of simple cores. Thus, in modern scientific computing, the potential for an algorithm to be *parallelized* is increasingly important. The potential speedup from parallelization is not infinite, and depending on the structure of the problem can sometimes be much less than expected. Amdahl's Law – the seemingly obvious point that the overall performance gained by optimizing a part of a program is limited by the amount that part gets used – can be written as

$$S = ((1 - p) + p/a)^{-1},$$

where S is the amount the program execution is sped up, p is the proportion of time the program spends in the section of code that is being optimized, and a is the factor by which the optimized section is accelerated. The quantity S might be less than you expect at first glance. For instance, if in your program you take a section of code that currently uses 75% of the computational time and (working some real magic!) make it 10 times faster, your program will only run 3 times faster than before. Even more extreme: if you take a section of code that currently uses 98% of the computational time and implement a strategy that makes it run 10^10 faster – blazing fast! – your program will still only run 50 times faster than before.

These arguments apply to speeding up serial sections of code, and they equally apply to accelerating a section of of a program by employing parallel computing resources. Because of these constraints, the architectural choice of an algorithm need not just be a trade-off between $O(N^2)$ and $O(N \log N)$; it can also be a trade-off between clever-but-sequential algorithm and a simpler one that can more effectively harness the power of parallel hardware. We'll explore these concepts – and how to think about what parts of your code are actually worth optimizing

[70]To be more precise, Big-O provides an *asymptotic upper bound*. Computer scientists also have different notation for, e.g., *tight bounds* ($\Theta(g(N))$) – where $f(N)$ is bounded both above and below by $g(N)$ – and lower bounds ($\Omega(g(N))$).

– in more detail later in the course, but it is an important part of the modern computational landscape to have in your head from the beginning.

7.3.2 Performance vs. Simplicity

When deciding which algorithm to use, there is often an important trade-off between performance and simplicity. By simplicity I often mean something about how complex the algorithm is to implement[71], and also how easy it is to establish the correctness of that implementation. The performance of an algorithm is often reasoned about in terms of its asymptotic scaling, although the constant factor hidden by the Big-O notation can be quite important. That constant factor is, itself, often directly correlated with the simplicity of the algorithm; depending on the problem size you are interested in, it might be absolutely crucial.

As an example of a problem where these considerations come up, consider the multiplication of two $N \times N$ matrices. I'm choosing this example because matrix multiplication is not just a textbook exercise; it is a fundamental operation at the heart of countless computational methods. A simpler example – perhaps different algorithms to sort a list – could be used to make the same point, but matrix multiplication is more interesting, and there are *open questions* at the cutting edge of algorithmic research about how fast matrix multiplication can actually be!

The simplest matrix multiplication algorithm is one that directly implements what you probably think of as the *definition* of matrix multiplication. As I'm sure I don't need to tell you, given matrices A and B, the i, j element of $\mathbf{C} = \mathbf{AB}$ is

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

Said another way, the i, j element of the product is the dot product of the ith row of $\mathbf{A}$ with the jth column of $\mathbf{B}$.

Assuming that the matrices in question don't have any special structure (they aren't the identity matrix, they aren't extremely sparse, etc), it is straightforward to analyze the asymptotic scaling of this approach to matrix multiplication: For every element in $\mathbf{C}$ we compute the dot product, which involves N multiplications and $N-1$ additions. Since there are N^2 elements to compute this for, we confidently say that naive matrix multiplication is $O(N^3)$. Implementing the algorithm corresponding to this is straightforward, and the correctness of that algorithm is easy to verify (and, indeed, extend to matrices whose shape is not square).

If you haven't heard of Strassen's algorithm [7] you might have concluded that matrix multiplcation *itself* is $O(N^3)$. Consider this product of 2×2 matrices:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

This is standard matrix multiplication, and we can see that it involves 8 total multiplications and 4 total additions – exactly as we computed above. Of course, I would have written down exactly the same formula if each of the elements above was itself a block matrix – in which case this is an equation with 8 total *matrix* multiplications and 4 total *matrix* additions.

[71] With the caveat that some programming languages or libraries make it quite simple to *use* algorithms that would be horrible to write from scratch.

Let's go ahead and define the following seven objects (they might be scalars, or they might themselves be block matrices):

$$\begin{aligned} I &= (A_{11} + A_{22})(B_{11} + B_{22}) & V &= (A_{11} + A_{12})B_{22} \\ II &= (A_{21} + A_{22})B_{11} & VI &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ III &= A_{11}(B_{12} - B_{22}) & VII &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ IV &= A_{22}(B_{21} - B_{11}) \end{aligned}$$

In terms of these, we can write the original matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} I + IV - V + VII & III + V \\ II + IV & I - II + III + VI \end{pmatrix}.$$

We have turned 8 total multiplications and four total additions into 7 total multiplications and 18 total additions. If these were scalars that would be a net loss, but for matrices it could be a massive win – naive matrix multiplication is $O(N^3)$ and matrix addition is $O(N^2)$, so reducing the number of matrix multiplications involved is great! The real trick comes from the idea of *recursively* applying this block decomposition over and over again; in the asymptotic limit this approach to multiplying matrices has complexity $O(N^{\log_2 7})$. This might not seem like much, but as $N \to \infty$ it is a huge win[72].

Clearly, though, actually implementing Strassen's approach is much more complex than implementing naive matrix multiplication, it requires extra memory to store the intermediate matrices, and for small matrices it actually involves *more* arithmetic operations! Thus, in this case there is a crossover in matrix scale beyond which Strassen's approach is faster and below which it is actually slower; the precise value of the crossover (usually estimated to be for dense matrices with thousands of rows and columns) is also a function of the *hardware* the algorithms are run on. The general lesson is that the choice of algorithm you employ should depend on the scale of the problem you intend to solve (perhaps with some consideration given to the *range of scales* you might ever consider).

7.4 Coda

The three components of project architecture discussed above are linked together. The *static* organization of our files – hopefully straightforward – provides a clean, version controlled home for our entire project. The *execution* architecture of our scripts determines how we carry out the reproducible process of our scientific explorations. Together, these two components govern the projects "health" – they determine how maintainable, extensible, and reproducible our project will be (and whether we will be able to understand any of it a week or a month after we work on it).

It is the *logical* architecture – the algorithms we choose and the way we weave them together – that casts the longest shadow. An inefficient algorithm cannot be saved by a tidy file structure, and a fundamentally slow or unstable numerical method cannot be fixed by a clever execution

[72]Strassen's approach is not even the best known asymptotic algorithm! At the moment the bound on how matrix multiplication intrinsically scales is bounded by something like $O(N^{2.371552})$ [8].

script. The logical design sets the hard limits on what is computationally feasible; this leads to some core principles for project planning.

Project planning for computational projects

1. **Identify the computational core:** In most projects, one or just a few parts of the code consumer the majority of the computational time – this might be the pairwise calculation of all forces, or the need for matrix inversion, or some geometric calculation on your data structures. Identify the "hot loops" first; your project's scope will be determined by how efficient the computational bottlenecks are.

2. **Design for change:** Your first idea is rarely your best, so build your code with a modular, high-level API that allows you to swap out the core components without having to demolish the entirety of your code.

3. **Optimize what matters:** Use Amdahl's Law as your guide. Before you spend a week optimizing a function, *measure its impact*. A 10x speedup on a part of the code that only accounts for 1% of the runtime is probably a waste. Not of your computer's resources, but of your own time.

Bibliography

[1] William Jones. *Synopsis Palmariorum Matheseos: Or, a New Introduction to the Mathematics*. J. Matthews for Jeff. Wale at the Angel in St. Paul's Church-Yard, 1706.

[2] William Oughtred. *Clavis Mathematicae denuo limita, sive potius fabricata*. Lichfield, 1631.

[3] Florian Cajori. *A history of mathematical notations*, volume 1. Courier Corporation, 1993.

[4] Berni J Alder and Thomas Everett Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[5] Gregorii Aleksandrovich Galperin. Playing pool with π (the number π from a billiard point of view). *Regular and chaotic dynamics*, 8(4):375–394, 2003.

[6] F Chiappetta, C Meringolo, P Riccardi, R Tucci, A Bruzzese, and G Prete. Boyle, huygens and the 'anomalous suspension'of water. *Physics Education*, 59(4):045026, 2024.

[7] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[8] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[9] Leonhard Euler. *Institutiones calculi integralis*, volume 1. Impensis Academiae Imperialis Scientiarum, 1768.

[10] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[11] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[12] John C Butcher. Implicit runge-kutta processes. *Mathematics of computation*, 18(85):50–64, 1964.