Eric C. Anderson

Practical Computing and Bioinformatics for Conservation and Evolutionary Genomics

Contents

List of Tables

List of Figures

Preface

This is a collection of blurbs Eric started writing in the last year to help remind himself and others of some useful things to know for bioinformatics.

It was started during an extended government shutdown, when it seemed like writing a book might be in order. Fortunately, the shutdown ended eventually. It is not clear whether a book will ever come of it, but these notes shall be up on the web indefinitely, so feel free to use them!

0

Introduction

Nothing here yet.

0

Eric's Notes of what he might do

This is where I am going to just throw out ideas and start to organize them. My thought was that while I am actually doing bioinformatics, etc. in my normal day-to-day work I will analyze what I am doing and figure out all the different tools that I am using and organize that or pedagogy.

- Note: I am going to make a companion repository called `mega-bioinf-pop-gen-examples` that will house all of the data sets and things for exercises.

0.1 Table of topics

Man! There is going to be a lot to get through. My current idea is to meet three times a week. The basic gist of those three sessions will be like this:

1. **Fundamental Tools / Environments**: I am thinking 5 weeks on Unix, 1 Week on HPC, 6 Weeks on R/Rstudio, and 3 on Python from within Rstudio (so that students know enough to run python modules like moments.)
2. **Theory and Background**: Population-genetic and bioinformatic theory. Alignment and BW transforms, the coalescent, Fst, etc. Basically things that are needed to understand (to some degree) what various programs/analyses are doing under the hood.
3. **Application and Practice**: Getting the students to get their feet wet and their fingers dirty actually doing it. This time should be entirely practical, with students doing an exercise (in pairs or groups, possibly) with me (or someone else, maybe CH) overseeing.

Week	Fundamental Tools	Theory and Background	Application and Practice
1	*Unix Intro*: filesystem; absolute and relative paths, everything is a file; readable, writable, executable; PATH; .bashrc — hack everyone's to get the time and directory; TAB-completion; `cd`, `ls` (colored output), `cat`, `head`, `less`; stdout and stderr and file redirection of either with `>` and `2>`; the `;` vs `&`. Using TextWrangler with `edit` and we need a PC equivalent…	*Data Formats*: fasta, fastq, SAM, BAM, VCF, BCF	Command line drills

Week	Fundamental Tools	Theory and Background	Application and Practice
2	Programs, binaries, compiling, installing, package management; software distribution; GitHub and sourceforge; admin privileges and sudo, and how you probably won't have that on a cluster.	Fundamental programming concepts; Scripts vs binaries (i.e. compiled vs interpreted languages); dependencies: headers and libraries; Modularization; Essential algorithms; compression;	samtools, vcftools, bcftools. hands on, doing stuff with them, reading the man pages, exercises.
3	*Programming on the shell*: variables and variable substitution; Globbing and path expansion; variable modifications; loops; conditionals;		
4	*sed, awk, and regular expressions*		
5	*HPC*: clusters; nodes; cores; threads. SGE and/or SLURM; `qsub`; `qdel`; `qacct`; `myjobs`; job arrays.		

Part I

Part I: Essential Computing Skills

1

Overview of Essential Computing Skills

What up with this? It appears that bookdown does not let you write things at the beginning of a Part without putting it under a chapter heading. Oh well.

2

Essential Unix/Linux Terminal Knowledge

Unix was developed at AT&T Bell Labs in the 1960s. Formally "UNIX" is a trademarked operating system, but when most people talk about "Unix" they are talking about the *shell*, which is the text-command-driven interface by which Unix users interact with the computer.

The Unix shell has been around, largely unchanged, for many decades because it is *awesome*. When you learn it, you aren't learning a fad, but, rather, a mode of interacting with your computer that has been time tested and will likely continue to be the lingua franca of large computer systems for many decades to come.

For bioinformatics, Unix is the tool of choice for a number of reasons: 1) complex analyses of data can be undertaken with a minimum of words; 2) Unix allows automation of tasks, especially ones that are repeated many times; 3) the standard set of Unix commands includes a number of tools for managing large files and for inspecting and manipulating text files; 4) multiple, successive analyses upon a single stream of data can be expressed and executed efficiently, typically without the need to write intermediate results to the disk; 5) Unix was developed when computers were extremely limited in terms of memory and speed. Accordingly, many Unix tools have been well optimized and are appropriate to the massive genomic data sets that can be taxing even for today's large, high performance computing systems; 6) virtually all state-of-the-art bioinformatic tools are tailored to run in a Unix environment; and finally, 7) essentially every high-performance computer cluster runs some variant of Unix, so if you are going to be using a cluster for your analyses (which is highly likely), then you have gotta know Unix!

2.1 Getting a bash shell on your system

A special part of the Unix operating system is the "shell." This is the system that interprets commands from the user. At times it behaves like an interpreted programming language, and it also has a number of features that help to minimize the amount of typing the user must do to complete any particular

tasks. There are a number of different "shells" that people use. We will focus on one called "bash," which stands for the "Bourne again shell." Many of the shells share a number of features.

Many common operating systems are built upon a Unix or upon Linux—an open-source flavor of Unix that is, in many scenarios, indistinguishable. Hereafter we will refer to both Unix and Linux as "Unix" systems). For example all Apple Macintosh computers are built on top of the Berkeley Standard Distribution of Unix and bash is the default shell. Many people these days use laptops that run a flavor of Linux like Ubuntu, Debian, or RedHat. Linux users should ensure that they are running the bash shell. This can be done by typing "bash" at the command line, or inserting that into their profile. To know what shell is currently running you can type:

```
echo $0
```

at the Unix command line. If you are running `bash` the result should be

```
-bash
```

PCs running Microsoft Windows are something of the exception in the computer world, in that they are not running an operating system built on Unix. However, Windows 10 now allows for a Linux Subsystem to be run. For Windows, it is also possible to install a lightweight implementation of bash (like Git Bash). This is helpful for learning how to use Unix, but it should be noted that most bioinformatic tools are still difficult to install on Windows.

2.2 Navigating the Unix filesystem

Most computer users will be familiar with the idea of saving documents into "folders." These folders are typically navigated using a "point-and-click" interface like that of the Finder in Mac OS X or the File Explorer in a Windows system. When working in a Unix shell, such a point-and-click interface is typically not available, and the first hurdle that new Unix users must surmount is learning to quickly navigate in the Unix filesystem from a terminal prompt. So, we begin our foray into Unix and its command prompt with this essential skill.

When you start a Unix shell in a terminal window you get a *command prompt* that might look something like this:

```
my-laptop:~ me$
```

or, perhaps something as simple as:

```
$
```

or maybe something like:

```
/~/--%
```

We will adopt the convention in this book that, unless we are intentionally doing something fancier, the Unix command prompt is given by a percent sign, and this will be used when displaying text typed at a command prompt, followed by output from the command. For example

```
% pwd
/Users/eriq
```

shows that I issued the Unix command `pwd`, which instructs the computer to **p**rint **w**orking **d**irectory, and the computer responded by printing `/Users/eriq`, which, on my Mac OS X system is my *home directory*. In Unix parlance, rather than speaking of "folders," we call them "directories;" however, the two are essentially the same thing. Every user on a Unix system has a home directory. It is the domain on a shared computer in which the user has privileges to create and delete files and do work. It is where most of your work will happen. When you are working in the Unix shell there is a notion of a *current working directory*—that is to say, a place within the hierarchy of directories where you are "currently working." This will become more concrete after we have encountered a few more concepts.

The specification `/Users/eriq` is what is known as an *absolute path*, as it provides the "address" of my home directory, `eriq`, on my laptop, starting from the *root* of the filesystem. Every Unix computer system has a root directory (you can think of it as the "top-most" directory in a hierarchy), and on every Unix system this root directory always has the special name, `/`. The address of a directory relative to the root is specified by starting with the root (`/`) and then naming each subsequent directory that you must go inside of in order to get to the destination, each separated by a `/`. For example, `/Users/eriq` tells us that we start at the root (`/`) and then we go into the `Users` directory (`Users`) and then, from there, into the `eriq` directory. Note that `/` is used to mean the root directory when at the beginning of an absolute path, but in the remainder of the path its meaning is different: it is used merely as a separator

between directories nested within one another. Figure 2.1 shows an example hierarchy of some of the directories that are found on the author's laptop.

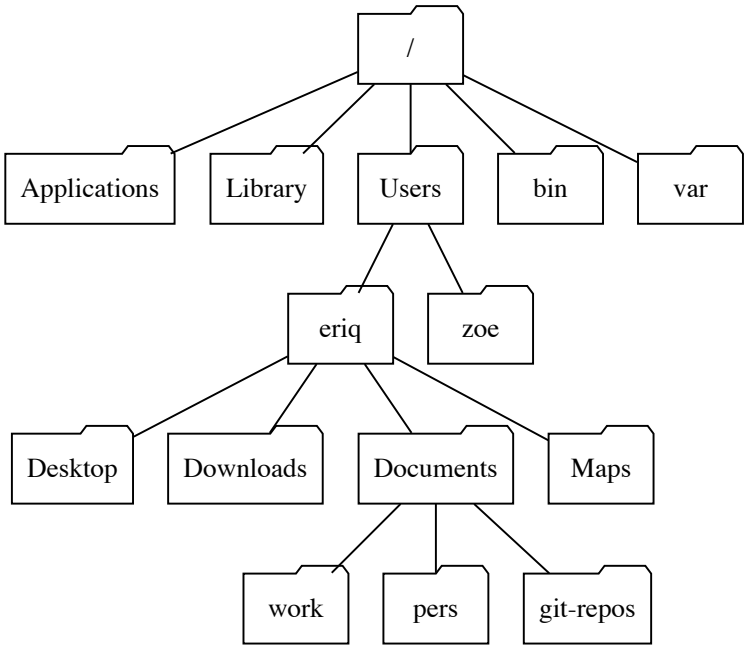


FIGURE 2.1: A partial view of the directories on the author's laptop.

From this perspective it is obvious that the directory `eriq` lives inside `Users`, and also that, for example, the absolute path of the directory `git-repos` would be `/Users/eriq/Documents/git-repos`.

Absolute paths give the precise location of a directory relative to the root of the filesystem, but it is not always convenient, nor appropriate, to work entirely with absolute paths. For one thing, directories that are deeply nested within many others can have long and unwieldy absolute path names that are hard to type and can be difficult to remember. Furthermore, as we will see later in this book, absolute paths are typically not *reproducible* from one computer's filesystem to another. Accordingly, it is more common to give the address of directories using *relative paths*. Relative paths work much like absolute paths; however, they do not start with a leading `/`, and hence they do not take as their starting point the root directory. Rather, their starting point is implicitly taken to be the current working directory. Thus, if the current

working directory is `/Users/eriq`, then the path `Documents/pers` is a relative path to the `pers` directory, as can again be seen in Figure 2.1.

The special relative path symbol `..` means "the directory that is one level higher up in the hierarchy." So, if the current working directory were `/Users/eriq/Documents/git-repos`, then the path `..` would mean `/Users/eriq/Documents`, the path `../work` gives the directory `/Users/eriq/Documents/work`, and, by using two or more `..` symbols separated by forward slashes, we can even go up multiple levels in the hierarchy: `../../../zoe` is a relative path for `/Users/zoe`, when the current working directory is `/Users/eriq/Documents/git-repos`.

When naming paths, another useful Unix shorthand is `~` (a tilde) which denotes the user's home directory. This is particularly useful since most of your time in a Unix filesystem will be spent in a directory within your home directory. Accordingly, `~/Documents/work` is a quick shorthand for `/Users/eriq/Documents/work`. This is essential practice if you are working on a large shared computing resource in which the absolute path to your home directory might be changed by the system administrator when restructuring the filesystem.

A useful piece of terminology: in any path, the "final" directory name is called the *basename* of the path. Hence the basename of `/Users/eriq/Documents/git-repos` is `git-repos`. And the basename of `../../Users` is `Users`.

2.2.1 Changing the working directory with `cd`

When you begin a Unix terminal session, the current working directory is set, by default, to your home directory. However, when you are doing bioinformatics or otherwise hacking on the command line, you will typically want to be "in another directory" (meaning you will want the current working directory set to some other directory). For this, Unix provides the `cd` command, which stands for **c**hange **d**irectory. The syntax is super simple:

cd *path*

where *path* is an absolute or a relative path. For example, to get to the `git-repos` directory from my home directory would require a simple command `cd Documents/git-repos`. Once there, I could change to my `Desktop` directory with `cd ../../Desktop`. Witness:

```
% pwd
/Users/eriq
```

```
% cd Documents/git-repos/
% pwd
/Users/eriq/Documents/git-repos
% cd ../../Desktop
% pwd
/Users/eriq/Desktop
```

Once you have used `cd`, the working directory of your current shell will remain the same no matter how many other commands you issue, until you invoke the `cd` command another time and change to a different directory.

If you give the `cd` command with no path specified, your working directory will be set to your home directory. This is super-handy if you have been exploring the levels of a Unix filesystem above your home directory and cannot remember how to get back to your home directory. Just remember that

```
% cd
```

will get you back home.

Another useful shortcut is to supply `-` (a hyphen) as the path to `cd`. This will change the working directory back to where you were before your last invocation of `cd`, and it will tell you which directory you have returned to. For example, if you start in `/Users/eriq/Documents/git-repos` and then `cd` to `/bin`, you can get back to `git-repos` with `cd -` like so:

```
% pwd
/Users/eriq/Documents/git-repos
% cd /bin/
% pwd
/bin
% cd -
/Users/eriq/Documents/git-repos
% pwd
/Users/eriq/Documents/git-repos
```

Note the output of `cd -` is the newly-returned-to current working directory.

2.2.2 Updating your command prompt

When you are buzzing around in your filesystem, it is often difficult to remember which directory you are in. You can always type `pwd` to figure that out,

but the bash shell also provides a way to print the current working directory *within your command prompt*.

For example, the command:

```
PS1='[\W]--% '
```

redefines the command prompt to be the basename of the current directory surrounded by brackets and followed by `--%`:

```
% pwd
/Users/eriq/Documents/git-repos
% PS1='[\W]--% '
[git-repos]--% cd ../
[Documents]--% cd ../
[~]--% cd ../
[Users]--%
```

This can make it considerably easier to keep track of where you are in your file system.

We will discuss later how to invoke this change automatically in every terminal session when we talk about customizing environments in Section 2.5.

2.2.3 TAB-completion for paths

Let's be frank…typing path names in order to change from one directory to another can feel awfully tedious, especially when your every neuron is screaming, "Why can't I just have a friggin' Finder window to navigate in!" Do not despair. This is a normal reaction when you first start using Unix. Fortunately, Unix file-system navigation can be made much less painful (or even enjoyable) for you by becoming a master of *TAB-completion*. Imagine the Unix shell is watching your every keystroke and trying to guess what you are about to type. If you type the first part of a directory name after a command like `cd` and then hit the TAB key, the shell will respond with its best guess of how you want to complete what you are typing.

Take the file hierarchy of Figure 2.1, and imagine that we are in the root directory. At that point, if we type `cd A`, the shell will think "Ooh! I'll bet they want to change into the directory `Applications` because that is the only directory that starts with `A`. Sure enough, if you hit TAB, the shell adds to the command line so that `cd A` becomes `cd Applications/` and the cursor is still waiting for further input at the end of the command. Boom! That was way easier (and more accurate) than typing all those letters after `A`.

Developing a lightning-fast TAB-completion trigger finger is, quite seriously, essential to surviving and thriving in Unix. Use your left pinky to hit TAB. Hone your skills. Make sure you can hit TAB with your eyes closed. TAB early and TAB often!

Once you can hit TAB instantly from within the middle of any phrase, you will also want to understand a few simple rules of TAB completion:

1. If you try TAB-completing a word on the command line that is not at the beginning of the command line (i.e., you are typing a word after a command like `cd`), then the shell tries to complete the word with a *directory name* or a *file name*.
2. The shell will only complete an *entire* directory or file name if the name *uniquely* matches the first part of the path that has been entered. In our example, there were no other directories than `Applications` in `/` that start with `A`, so the shell was certain that we must have been going for `Applications`.
3. If there is more than one directory or file name that matches what you have already typed, then, the first time you hit TAB, nothing happens, but the *second* time you hit TAB, the shell will print a list of names that match what you have written so far. For example, in our Figure 2.1 example, hitting TAB after typing `cd ~/D` does nothing. But the second time we hit TAB we get a list of matching names:

```
% cd ~/D
Desktop/  Documents/  Downloads/
```

   So, if we are heading to `Documents` we can see that adding `oc` to our command line, to create `cd Doc` would be sufficient to allow the shell to uniquely and correctly guess where we are heading. `cd Doc` will TAB-complete into `cd Documents/`
4. If there are multiple directory or file names that match the current command line, and they share more letters than those currently on the command line, TAB-completion will complete the name to the end of the shared portion of the name. An example helps: let's say I have the following two directories with hideously long names in my `Downloads` folder:

```
WIFL.rep_indiv_est.mixture_collection.count.gr8-results
WIFL.rep_indiv_est.mixture_collection.count-results
```

Then, TAB completing on `~/Downloads/WIFL.rep` will partially complete so that the prompt and command look like:

```
% cd ~/Downloads/WIFL.rep_indiv_est.mixture_collection.count
```

and hitting TAB twice gives:

```
% cd ~/Downloads/WIFL.rep_indiv_est.mixture_collection.count
WIFL.rep_indiv_est.mixture_collection.count-results
WIFL.rep_indiv_est.mixture_collection.count.gr8-results
```

At this point, adding `-` and TAB completing will give the first of those directories.

The last example shows just how much typing TAB completion can save you. So, don't be shy about hitting that TAB key. When navigating your filesystem (or writing longer command lines that require paths of files) you should consider hitting TAB after every 1 or 2 letters. In routine work on the command line, probably somewhere around 25% or more of my keystrokes are TABs. Furthermore, a TAB is never going to execute a command, and it typically won't complete to a path that you don't want (unless you got the first part of its name wrong), so there isn't any risk to hitting TAB all the time.

2.2.4 Listing the contents of a directory with `ls`

So far we have been focusing mostly on directories. However, directories themselves are not particularly interesting—they are merely containers. It is the *files* inside of directories that we typically work on. The command `ls` lists the contents—typically files or other directories—within a directory.

Invoking the `ls` command without any other arguments (without anything after it) returns the contents of the current working directory. In our example, if we are in `/Users` then we get:

```
% ls
eriq  zoe
```

By default, `ls` gives output in several columns of text, with the directory contents sorted lexicographically. For example, the following is output from the `ls` command in a directory on a remote Unix machine:

```
% ls
bam                        map-sliced-fastqs-etc.sh
bam-slices                 play
bwa-run-list.txt           REDOS-map-sliced-fastqs-etc.sh
fastq-file-prefixes.txt    sliced
fqslice-22.error           slice-fastqs.sh
fqslice-22.log             slicer-lines.txt
map-etc.sh                 Slicer-Logs-summary.txt
```

The first line shows the command prompt and the command: `% ls`, and the remainder is the output of the command.

Invoked without any further arguments, the `ls` command simply lists the contents of the current working directory. However, you can also direct `ls` to list the contents of another directory by simply adding the path (absolute or relative) of that directory on the command line. For example, continuing with the example in Figure 2.1, when we are in the home directory (`eriq`) we can see the directories/files contained within `Documents` like so:

```
[~]--% ls Documents
git-repos/   pers/   work/
```

If you give paths to more than one directory as arguments to `ls`, then the contents of each directory are listed after a heading line that gives the directory's path (as given as an argument to `ls`), followed by a colon. For example:

```
[~]--% ls Documents/git-repos Documents/work
Documents/git-repos:
ARCHIVED_mega-bioinf-pop-gen.zip  lowergranite_0.0.1.tar.gz
AssignmentAdustment/              mega-bioinf-pop-gen-examples/
CKMRsim/                          microhaps_np/

Documents/work:
assist/          maps/            oxford/          uw_days/
courses_audited/ misc/            personnel/
```

You might also note in the above example, that some of the paths listed within each of the two directories are followed by a slash, `/`. This `ls` customization denotes that they are directories themselves. Much like your command prompt, `ls` can be customized in ways that make its output more informative. We will return to that in Section 2.5.

If you pass the path of a file to `ls`, and that file exists in your filesystem, then `ls` will respond by printing the file's path:

```
% ls Documents/git-repos/lowergranite_0.0.1.tar.gz
Documents/git-repos/lowergranite_0.0.1.tar.gz
```

If the file does not exist you get an error message to that effect:

```
% ls Documents/try-this-name
ls: Documents/try-this-name: No such file or directory
```

The multi-column, default output of `ls` is useful when you want to scan the contents of a directory, and quickly see as many files as possible in the fewest lines of output. However, this output format is not well structured. For example, you don't know how many columns are going to be used in the default output of `ls` (that depends on the length of the filenames and the width of your terminal), and it offers little information beyond the names of the files.

You can tell the `ls` command to provide more information, by using it with the `-l` option. Appropriately, with the `-l` option, the `ls` command will return output in *long* format:

```
2019-02-08 21:09 /osu-chinook/--% ls -l
total 108
drwxr-xr-x  2 eriq kruegg  4096 Feb  7 08:26 bam
drwxr-xr-x 14 eriq kruegg  4096 Feb  8 15:56 bam-slices
-rw-r--r--  1 eriq kruegg 17114 Feb  7 20:16 bwa-run-list.txt
-rw-r--r--  1 eriq kruegg   824 Feb  6 14:14 fastq-file-prefixes.txt
-rw-r--r--  1 eriq kruegg     0 Feb  7 20:14 fqslice-22.error
-rw-r--r--  1 eriq kruegg     0 Feb  7 20:14 fqslice-22.log
-rwxr--r--  1 eriq kruegg  1012 Feb  7 07:59 map-etc.sh
-rwxr--r--  1 eriq kruegg  1138 Feb  7 20:56 map-sliced-fastqs-etc.sh
drwxr-xr-x  3 eriq kruegg  4096 Feb  7 13:01 play
-rwxr--r--  1 eriq kruegg  1157 Feb  8 15:08 REDOS-map-sliced-fastqs-etc.sh
drwxr-xr-x 14 eriq kruegg  4096 Feb  8 15:49 sliced
-rwxr--r--  1 eriq kruegg   826 Feb  7 20:09 slice-fastqs.sh
-rw-r--r--  1 eriq kruegg  1729 Feb  7 16:11 slicer-lines.txt
```

Each row contains information about only a single file. The first column indicates what kind of file each entry is, and also tells us which users have permission to do certain things with the file (more on this in a few sections). The third and fourth columns show that the owner of each file is `eriq`, who is a user in the group called `kruegg`. After that is the size of the file (in bytes) and the date and time it was last modified.

There are a few options to `ls` that are particularly useful. One is `-a`, which causes `ls` to include in its listing all files, even *hidden* ones. In a Unix file

system, any file whose name starts with a `.` is considered a *hidden* file. Commonly, such files are configuration files or other files used by programs that you typically don't interact with directly. (We will see an example of this when we start working with `git` for version control, Section 9.2.) The `-d` option for `ls` is also' quite handy. Recall that when you provide the name of a directory as an argument to `ls`, the default behavior is to list the contents of the directory. This can be troublesome when you are listing the contents of a subdirectory: `ls ~/Documents/git-repos/*` lists the contents (which can be substantial) of each of the directories in my directory, but I might only want to know the name of each of those directories, rather than their full contents. `ls -d ~/Documents/git-repos` will do that for you. Finally, the `-R` option to `ls` will cause the operating system to drill down, *recursively* into all the subdirectories of the one you supplied to the command, and list their contents, as well.

2.2.5 Globbing

If you have ever had to move a large number of files of a certain type from one folder to another in a Finder window, you know that individually clicking and selecting each one and then dragging them can be a tedious task (not to mention the disaster that ensues if you slip on your mouse and end up dropping all the files some place you did not intend). Unix provides a wonderful system called *filename expansion* or "globbing" for quickly providing the names of a large number of files and paths which let's you operate on multiple files quickly and efficiently. In short, globbing allows for *wildcard matching* in path names. This means that you can specify multiple files that have names that share a common part, but differ in other parts.

The most widely used (and the most permissive) wildcard is the asterisk, `*`. It matches anything in a file name. So, for example:

- `*.vcf` will expand to any files in the current directory with the suffix `.vcf`.
- `D*s` will expand to any files that start with an uppercase `D` and end with an `s`.
- `*output-*.txt` will expand to any files that include the phrase `output-` somewhere in their name and also end with `.txt`.
- `*` will expand to all files in the current working directory.
- `/usr/local/*/*.sh` will expand to any files ending in `.sh` that reside within any directory that is within the `/usr/local` directory.

Actually, there is some arcana here: Names of files or directories that start with a dot (a period) will not expand unless the dot is included explicitly. Files with names starting with a dot are "hidden" files in Unix. You

also will not see them in the results of `ls`, unless you use the `-a` option: `ls -a`.

After the asterisk, the next most commonly-used wildcard is the question mark, `?`. The question mark denotes any single character in a file name. For example. If you had a series of files that looked like `AA-file.txt`, `AB-file.txt`, …, `AZ-file.txt`. You could get get all those by using `A?-file.txt`. This would not expand to, for example, `AAZ-file.txt`, if that were in the directory.

You can be more specific in globbing by putting things within `[` and `]`. For example: `A[A-D]*` would pick out any files starting with, `AA`, `AB`, `AC`, or `AD`. Or you could have said `A[a-d]*` which would get any files starting with `Aa`, `Ab`, `Ac`, or `Ad`. And you can also do it with numbers: `[0-9]`. You can also negate the contents of the `[]`, with `^`. Thus, `100_[^ABC]*` picks out all files that start with `100_` followed by anything that is *not* and `A`, `B`, or a `C`.

Finally, you can be really specific about replacements in file names by iterating over different possibilities with a comma-separated list within curly braces. For example, `img.{png,jpg,svg}` will iterate over the values in curly braces and expand to `img.png img.jpg img.svg`. Interestingly, with curly braces, this forms all those file names whether they exist or not. So, unlike `*` it isn't really matching available file names.

The last thing to note about all of these globbing constructs is that they are not intimately associated with the `ls` command. Rather, they simply provide expansions on the command line, and the the `ls` command is listing all those files. For example, try `echo *.txt`.

2.2.6 What makes a good file-name?

If the foregoing discussion suggests to you that it might not be good to use an actual `*`, `?`, `[`, or `{` in names that you give to files and directories on your Unix system, then congratulations on your intuition! Although you can use such characters in your filenames, they have to be preceded by a backslash, and it gets to be a huge hassle. So don't use them in your file names. Additionally, characters such as `#`, `|`, and `:` do not play well for file names. Don't use them!

Another pet peeve of mine (and anyone who uses Unix) are file names that have spaces in them. In Windows and on a Mac it is easy to create file names that have spaces in them. In fact, the standard Windows system comes with such space-containing directory names as `My Documents` or `My Pictures`. Yikes! Please *don't ever do that in your Unix life!* One can deal with spaces in file names, but there is really no reason to include spaces in your file names, and having spaces in file names will typically break a good many scripts. Rather

than a space, use an underscore, `_`, or a dash, `-`. You've gotta admit that, not only does `My-Documents` work better, but it actually looks better too!

However, should you have to deal with files having spaces in their name, you can address them by either backslash escaping the spaces, or putting the whole file name in quotation marks (single or double quotation marks will work). If you have a file called `dumb file name.jpg`, you can address it on the command line as either of the following three:

```
dumb\ file\ name.jpg
"dumb file name.jpg"
'dumb file name.jpg'
```

To make your life easier, however, the bottom line is that you should name your files on a Unix system using only upper- and lowercase letters (Unix file systems are case-sensitive), numerals, and the following three punctuation characters: `.`, `-`, and `_`. Though you can use other punctuation characters, they often require special treatment, and it is better to avoid them altogether.

2.3 The anatomy of a Unix command

Nearly every Unix command that you might invoke follows a certain pattern. First comes the `command` itself. This is the word that tells the system the name of the command that you are actually trying to do. After that, often, you will provide a series of *options* that will modify the behavior of the command (for example, as we have seen, `-l` is an option to the `ls` command). Finally, you might then provide some *arguments* to the functions. These are typically paths to files or directories that you would like the command to operate on. So, in short, a typical Unix command invocation will look like this:

`command` *options arguments*

Of course, there are exceptions. For example, when invoking Java-based programs from your shell, arguments might be supplied in ways that make them look like options, etc. But, for the most part, the above is a useful way of thinking about Unix commands.

Sometimes, especially when using `samtools` or `bcftools`, the `command` part of the command line might including a command and a subcommand, like `samtools view` or `bcftools query`. This means that the operating system is calling the program `samtools` (for example), and then samtools interprets the next token (`view`) to know that it needs to run the `view` routine, and interpret all following options in that context.

We will now break down each element in "`command` *options arguments*".

2.3.1 The `command`

When you type a command at the Unix prompt, whether it is a command like `ls` or one like `samtools` (Section **??**), the Unix system has to search around the filesystem for a file that matches the command name and which provides the actual instructions (the computer code, if you will) for what the command will actually do. It cannot be stressed enough how important it is to understand where and how the bash shell searches for these command files. Understanding this well, and knowing how to add directories that the shell searches for executable commands will alleviate a lot of frustration that often arises with Unix.

In brief, all Unix shells (and the bash shell specifically) maintain what is called an *environment variable* called `PATH` that is a colon-separated list of pathnames where the shell searches for commands. You can print the `PATH` variable using the `echo` command:

```
echo $PATH
```

On a freshly installed system without many customizations the `PATH` might look like:

```
/usr/bin:/bin:/usr/sbin:/sbin
```

which is telling us that, when bash is searching for a command, it searches for a file of the same name as the command first in the directory `/usr/bin`. If it finds it there, then it uses the contents of that file to invoke the command. If it doesn't find it there, then it next searches for the file in directory `/bin`. If it's not there, it searches in `/usr/sbin`, and finally in `/sbin`. If it does not find the command in any of those directories then it returns the error `command not found`.

When you install programs on your own computer system, quite often the installer will modify a system file that specifies the `PATH` variable upon startup. Thus after installing some programs that use the command line on a Mac system, the "default" `PATH` might look like:

```
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Library/TeX/texbin
```

2.3.2 The *options*

Sometimes these are called flags, and they provide a convenient way of telling a Unix command how to operate. We have already seen a few of them, like the `-a`, `-l` or `-d` options to `ls`.

Most, but not all, Unix tools follow the convention that options specified by a single letter follow a single dash, while those specified by multiple letters follow two dashes. Thus, the `tar` command takes the single character options `-x`, `-v`, and `-f`, but also takes an option named like `--check-links`. Some utilities also have two different names—a single-letter name and a long name—for many options. For example, the `bcftools view` program uses either `-a` or `--trim-alt-alleles` to invoke the option that trims alternate alleles not seen in a given subset of individuals. Other tools, like BEAGLE, are perfectly have options that are named with multiple letters following just a single dash.

Sometimes options take parameter values, like `bcftools view -g het`. In that case, `het` is a parameter value. Sometimes the parameter values are added to the option with an equals-sign.

With some unix utilities' single-letter options can be bunged together following a single dash, like, `tar -xvf` being synonymous with `tar -x -v -f`. This is not universal, and it is not recommended to expect it.

Holy Cow! This is not terribly standardized, and probably won't make sense till you really get in there and starting playing around in Unix…

2.3.3 Arguments

These are often file names, or other things that are not preceded by an option flag. For example, in the `ls` command:

```
ls -lrt dir3
```

`-lrt` is giving `ls` the options `-l`, `-r`, and `-t` and `dir3` is the *argument*—the name of the directory whose contents you should list.

2.3.4 Getting information about Unix commands

Zheesh! The above looks like a horrible mish-mash. How do we find out how to use/invoke different commands and programs in Unix? Well, most programs are documented, and you have to learn how to read the documentation.

If a utility is properly installed, you should be able to find its manual page with the `man` command. For example, `man ls` or `man tar`. These "man-pages",

as the results are called, have a fairly uniform format. They start with a summary of what the utility does, then then show how it is invoked and what the possible options are by showing a skeleton in the form:
"`command` *options arguments*"
and usually square brackets are put around things that are not required. This format can get quite ugly and hard to parse for an old human brain, like mine, but stick with it.

If you don't have a man-page for a program, you might try invoking the program with the `--help` option, or maybe with no option at all.

2.4 Handling, Manipulating, and Viewing files and streams

In Unix, there are two main types of files: *regular files* which are things like text files, figures, etc.—Anything that holds data of some sort. And then there are "special" files, which include *directories* which you've already seen, and *symbolic links* which we will talk about later.

2.4.1 Creating new directories

You can make a new directory with:

```
mkdir path
```

where `path` is a path specification (either absolute or relative). Note that if you want to make a directory within a subdirectory that does currently not exist, for example:

```
mkdir new-dir/under-new-dir
```

when `new-dir` does not already exist, then you have to either create `new-dir` first, like:

```
mkdir new-dir
mdkir new-dir/under-new-dir
```

or you have to use the `-p` option of `mkdir`, which creates all necessary parent directories as well, like:

```
mkdir -p new-dir/under-new-dir
```

If there is already a file (regular or directory) with the same path specifiation as a directory you are trying to create, you will get an error from `mkdir`.

2.4.2 Fundamental file-handling commands

For the day-to-day business of moving, copying, or removing files in the file system, the three main Unix commands are:

- `mv` for moving files and directories
- `cp` for copying files and directories
- `rm` for removing files and directories

These obviously do different things, but their syntax is somewhat similar.

2.4.2.1 `mv`

`mv` can be invoked with just two arguments like:

```
mv this there
```

which moves the file (or directory) from the path `this` to the path `there`.

- If `this` is a regular file (i.e. not a directory), and:
  - `there` is a directory,`this` gets moved inside of `there`.
  - `there` is a regular file that exists, then `there` will get overwritten, becoming a regular file that holds the contents of `this`.
  - `there` does not exist, it will be created as regular file whose contents are identical to those of `this`.
- If `this` is a directory and:
  - `there` does not exist in the filesystem, the directory `there` will be made and its contents will be the (former) contents of `this`
  - if `there` already exists, and is a directory, then the directory `this` will be moved inside of the directory `there` (i.e. it will become `there/this`).
  - if `there` already exists, but is not a directory, then nothing will change in the filesystem, but an an error will be reported. In all cases, whatever used to exist at path `this` will no longer be found there.

And `mv` can be invoked with multiple arguments, in which case the last one must be a directory *that already exists* that receives all the earlier arguments inside it. So, if you already have a directory named `dest_dir` then you can move a lot of things into it like:

```
mv file1 file2 dir1 dir2 dest_dir
```

You can also write that as as

```
mv file1 file2 dir1 dir2 dest_dir/
```

which makes its meaning a little more clear, but there is no requirement that the final argument have a trailing `/`.

Note, if any files in `dest_dir` have the same name as the files you are moving into `dest_dir` they *will* get overwritten.

So, you have gotta be careful not to overwrite stuff that you don't want to overwrite.

2.4.2.2 `cp`

This works much the same way as `mv` with two different flavors:

```
cp this there
```

and

```
cp file1 file2 dest_dir
# or
cp file1 file2 dest_dir/
```

The result is very much like that of `mv`, but instead of moving the file from one place to another (an operation that can actually be done without moving the data within the file to a different place on the hard drive), the `cp` command actually makes a full copy of files. Note that, if the files are large, this can take a long time.

2.4.2.3 `rm`

Finally we get to the very spooky `rm` command, which is short for "remove." If you say "rm myfile.txt" the OS will remove that file from your hard drive's directory. The data that were in the file might live on for some time on your hard drive—in other words, by default, `rm` does not wipe the file off your hard drive, but simply "forgets" where to look for that file. And the space that file took up on your hard drive is no longer reserved, and could easily be overwritten the next time you write something to disk. (Nonetheless, if you do `rm` a file, you should never expect to be able to get it back). So, be very careful

about using `rm`. It takes an `-r` option for recursively removing directories *and* all of their contents.

When used in conjunction with globbing, `rm` can be very useful. For example, if you wanted to remove all the files in a directory with a `.jpg` extension, you would do `rm *.jpg` from within that directory. However, it's a disaster to accidentally remove a number of files you might not have wanted to. So, especially as you are getting familiar with Unix, it is worth it to experiment with your globbing using `ls` first, to see what the results are, and only when you are convinced that you won't remove any files you really want should you end up using `rm` to remove those files.

2.4.3 "Viewing" Files

In a typical GUI-based environment, when you interact with files on your computer, you typically open the files with some application. For example, you open Word files with Microsoft Word. When working on the Unix shell, that same paradigm does not really exist. Rather, (apart from a few cases like the text editors, `nano`, `vim` and `emacs`, instead of opening a file and letting the user interact with it the shell is much happier just streaming the contents of the file to the terminal.

The most basic of such commands is the `cat` command, which *catenates* the contents of a file into a very special *data stream* called *stdout*, which is short for "standard output." If you don't provide any other instruction, data that gets streamed to *stdout* just shoots by on your terminal screen. If the file is very large, it might do this for a long time. If the file is a *text file* then the data in it can be written out in letters that are recognizable. If it is a *binary file* then there is no good way to represent the contents as text letters, and your screen will be filled with all sorts of crazy looking characters.

It is generally best not to `cat` very large files, especially binary ones. If you do and you need to stop the command from continuing to spew stuff across your screen, you can type `cntrl-c` which is the universal Unix command for "kill the current process happening on the shell." Usually that will stop it.

A note regarding terminals: On a Mac, the Terminal app is quite fast at spewing text across the screen. Megabytes of text or binary gibberish can flash by in seconds flat. This is not the case with the terminal window within RStudio, which can by abysmally slow, and usually doesn't store many lines of output.

Sometimes you want to just look at the top of a file. The `head` command shows you the first 10 lines of a file. That is valuable. The `less` command

shows a file one screenful at a time. You can hit the space bar to see the next screenful, and you can hit `q` to quit viewing the file.

Try navigating to a file and using `cat`, `head`, and `less` on it.

One particularly cool thing about `cat` is that if you say

```
cat file1 file2
```

it will catenate the contents of both files, in order, to *stdout*.

Now, one Big Important Unix fact is that many programs written to run in the Unix shell behave in the same way regarding their output: they write their output to *stdout*. We have already seen this with `ls`: its output just gets written to the screen, which is where *stdout* goes by default.

2.4.4 Redirecting standard output: > and >>

Unix starts to get really fun when you realize that you can "redirect" the contents of *stdout* from any command (or group of commands...see the next chapter!) to a file. To do that, you merely follow the command (and all its options and arguments) with `> path` where `path` is the path specifying the file into which you wish to redirect *stdout*.

Witness, try this:

```
# echo three lines of text to a file in the /tmp directory
echo "bing
bong
boing" > /tmp/file1

# echo three more lines of text to another file
echo "foo
bar
baz" > /tmp/file2

# now view the contents of the first file
cat /tmp/file1

# and the second file:
cat /tmp/file2
```

It is important to realize that when you redirect output into a file with `>`, any contents that previously existed in that file will be deleted (wiped out!). So be

careful about redirecting. Don't accidentally redirect output into a file that has valuable data in it.

The `>>` redirection operator does not delete the destination file before it redirects output into it. Rather, `>> file` means "append *stdout* to the contents that already exist in `file`." This can be very useful sometimes.

2.4.5 stdin, < and |

Not only do most Unix-based programs deliver output to standard output, but most utilities can also receive input from a file stream called *stdin* which is short for "standard input."

If you have data in a file that you want to send into standard input for a utility, you can use the `<` like this:

```
command < file
```

But, since most Unix utilities also let you specify the file as an argument, this is not used very much.

However, what is used all the time in Unix, and it is one of the things that makes it super fun, is the pipe, `|`, which says, "take *stdout* coming out of the command on the left and redirect it into *stdin* going into the command on the right of the pipe.

For example, if I wanted to count the number of files and directories stored in my `git-repos` directory, I could do

```
% ls -dl Documents/git-repos/* | wc
     174    1566   14657
```

which pipes the output of `ls -dl` (one line per file) into the *stdin* for the `wc` command, which counts the number of lines, words, and letters sent to its standard input. So, the output tells me that there are 174 files and directories in my directory `Documents/git-repos`.

Note that pipes and redirects can be combined in sequence over multiple operations or commands. This is what gives rise to the terminology of making "Unix pipelines:" the data are like streams of water coming into or out of different commands, and the pipes hook up all those streams into a pipeline.

2.4.6 stderr

While output from Unix commands is often written to *stdout*, if anything goes wrong with a program, then messages about that get written to a different stream called *stderr*, which, you guessed it! is short for "standard error". By default, both *stdout* and *stderr* get written to the terminal, which is why it can be hard for beginners to think of them as separate streams.
But, indeed, they are. Redirecting *stdout* with `>`, that does **not** redirect *stderr*.

For example. See what happens when we ask `ls` to list a file that does not exist:

```
[~]--% ls file-not-here.txt
ls: file-not-here.txt: No such file or directory
```

The error message comes back to the screen. If you redirect the output it still comes back to the screen!

```
[~]--% ls file-not-here.txt > out.txt
ls: file-not-here.txt: No such file or directory
```

If you want to redirect *stderr*, then you need to specify which stream it is. On all Unix systems, *stderr* is stream #2, so the `2>` syntax can be used:

```
[~]--% ls file-not-here.txt 2> out.txt
```

Then there is no output of *stderr* to the terminal, and when you `cat` the output file, you see that it went there!

```
[~]--% cat out.txt
ls: file-not-here.txt: No such file or directory
```

Doing bioinformatics, you will find that there will be failures of various programs. It is essential when you write bioinformatic pipelines to redirect *stderr* to a file so that you can go back, after the fact, to sleuth out why the failure occurred. Additionally, some bioinformatic programs write things like progress messages to *stderr* so it is important to know how to redirect those as well.

2.4.7 Symbolic links

Besides regular files and directories, a third type of file in Unix is called a *symbolic link*. It is a special type of file whose contents are just an absolute or

a relative path to another file. You can think of symbolic links as "shortcuts" to different locations in your file system. There are many useful applications of symbolic links.

Symbolic links are made using the `ln` command with the `-s` option. For example, if I did this in my home directory:

```
[~]--% ln -s /Users/eriq/Documents/git-repos/srsStuff srs
```

then `srs` becomes a file whose full listing (from `ls -l srs`) looks like:

```
lrwxrwxr-x  1 eriq  staff    40B Jan  9 19:24 srs@ -> /Users/eriq/Documents/git-repos/srsS
```

2.4.8 File Permissions

Unix systems often host many different users. Some users might belong to the same research group, and might like to be able to read the files (and/or use the programs) that their colleagues have in their accounts.

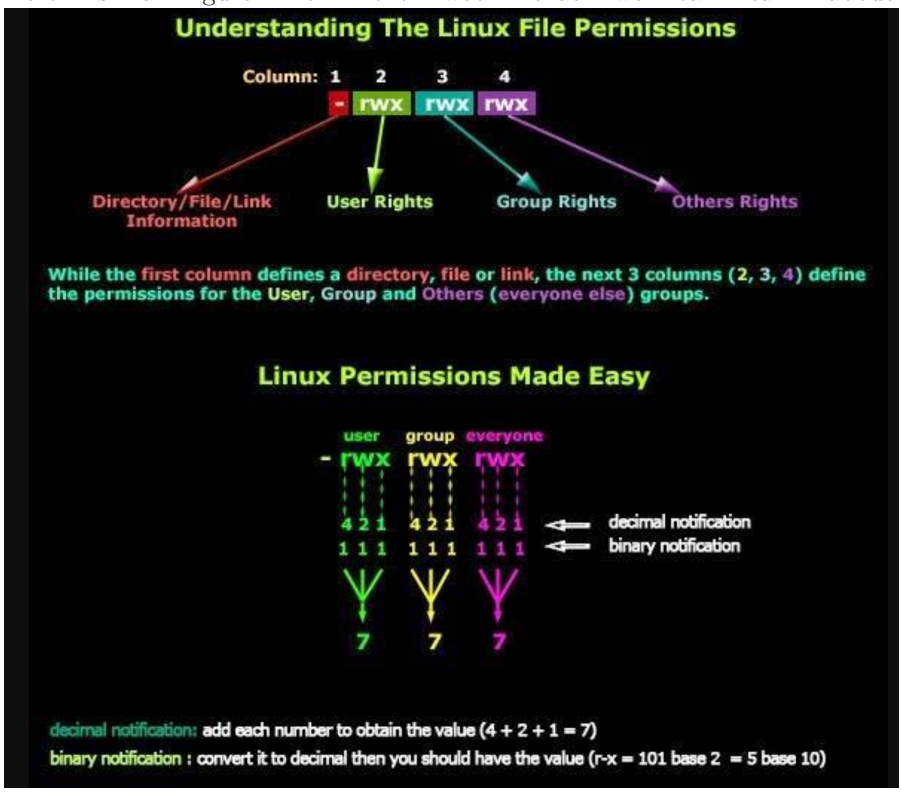
The Unix file system uses a system of permissions that gives rights to various classes of users to read, write, or execute files. The permissions associated with a file can be viewed using `ls -l`. They are captured in the first column which might look something like `-rwxr-xr-x`. When you first start looking at these, they can be distressingly difficult to visually parse. But you will get better at it! Let's start breaking it down now.

The file description string, in a standard Unix setting, consists of 10 characters.

- The first tells what kind of file it is: `-` = regular file, `d` = directory, `l` = symbolic link.
- The next group of three characters denote whether the owner/user of the file has permission to either read, write, or execute the file.

- The following two groups of three characters are the same thing for users within the users group, and for all other users, respectively.

Here is a figure from the web[1] that we can talk about:



Permissions can be changed with the `chmod` command. We will talk in class about how to use it with the octal representation of permissions.

2.4.9 Editing text files at the terminal

Sometimes you might need to edit text files at the command line.

The easiest text editor to use on the command line is `nano`. Try typing `nano a-file.txt` and type a few things. It is pretty self explanatory.

[1] https://unix.stackexchange.com/questions/183994/understanding-unix-permissions-and-file-types

2.5 Customizing your Environment

Previously we saw how to modify your command prompt to tell you what the current working directory is (remember `PS1='[\W]--% '`). The limitation of giving that command on the command line is that if you logout and then log back in again, or open a new Terminal window, you will have to reissue that command in order to achieve the desired look of your command prompt. Quite often a Unix user would like to make a number of customization to the look, feel, and behavior of their Unix shell. The bash shell allows these customization to be specified in two different files that are read by the system so as to invoke the customization. The two files are hidden files in the home directory: `~/.bashrc` and `~/.bash_profile`. They are used by the Unix system in two slightly different contexts, but for most purposes, you, the user, will not need or even want to distinguish between the different contexts. Managing two separate files of customization is unnecessary and requires duplication of your efforts, and can lead to inconsistent and confusing results, so here is what we will do:

1. Keep all of our customization in `~/.bashrc`.
2. Insert commands in `~/.bash_profile` that say, "Hey computer! If you are looking for customization in here, don't bother, just get them straight out of `~/.bashrc`.

We take care of #2, by creating the file `~/.bash_profile` to have the following lines in it:

```
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

Taking care of #1 is now just a matter of writing commands into `~/.bashrc`. In the following are some recommended customization.

2.5.1 Appearances matter

Some customization just change the way your shell looks or what type of output is given from different commands. Here are some lines to add to your `~/.bashrc` along with some discussion of each.

```
export PS1='[\W]--% '
```

This gives a tidier and more informative command prompt. The `export` command before it tells the system to pass the value of this *environment variable*, `PS1`, along to any other shells that get spawned by the current one.

```
alias ls='ls -GFh'
```

This makes it so that each time you invoke the `ls` command, you do so with the options `-G`, `-F`, and `-h`. To find out on your own what those options do, you can type `man ls` at the command line and read the output, but briefly: `-G` causes directories and different file types to be printed in different colors, `-F` causes a `/` to be printed after directory names, and other characters to be printed at the end of the names of different file types, and `-h` causes file sizes to be printed in an easily human-readable form when using the `-l` option.

2.5.2 Where are my programs/commands at?!

We saw in Section 2.3.1 that bash searches the directories listed in the `PATH` variable to find commands and executables. You can modify the PATH variable to include directories where you have installed different programs. In doing so, you want to make sure that you don't lose any of the other directories in `PATH`, so there is a certain way to go about redefining `PATH`. If you want to add the path `/a-new/program/directory` to your PATH variable you do it like this:

```
PATH=$PATH:/a-new/program/directory
```

2.6 A Few More Important Keystrokes

If a command "gets stuck" or is running longer than it should be, you can usually kill/quit it by doing `cntrl-c`.

Once you have given a command, it gets stored in your bash history. You can use the up-arrow key to cycle backward through different commands in your history. This is particularly useful if you are building up complex pipelines on the command line piece by piece, looking at the output of each to make sure

it is correct. Rather than re-typing what you did for the last command line, you just up-arrow it.

Once you have done an up-arrow or two, you can cycle back down through your history with a down-arrow.

Finally, you can search through your bash history by typing `cntrl-r` and then typing the word/command you are looking for. For example, if, 100 command lines back, you used a command that involved the program `awk`, you can search for that by typing `cntrl-r` and then typing `awk`.

One last big thing to note: the `#` is considered a *comment character* in bash. This means that any text following a `#` (unless it is backslash-escaped or inside quotation marks), until the next line ending, will be ignored by the shell.

2.7 A short list of additional useful commands.

Everyone should be familiar with the following commands, and the options that follow them on each line below. One might even think of scanning the manual page for each of these:

- `echo`
- `cat`
- `head`, `-n, -c`
- `tail`, `-n`
- `less`
- `sort`, `-n -b -k`
- `paste`
- `cut`, `-d`
- `tar`, `-cvf, -xvf`
- `gzip`, `-c`
- `du`, `-h -C`,
- `wc`
- `date`
- `uniq`
- `chmod`, `u+x`, `ug+x`
- `grep`

2.8 Two important computing concepts

2.8.1 Compression

Most file storage types (like text files) are a bit wasteful in terms of file space: every character in a text file takes the same number of bytes to store, whether it is a character that is used a lot, like `s` or `e`, or whether it is a character that is seldom seen in many text files, like `^`. *Compression* is the art of creating a code for different types of data that uses fewer bits to encode "letters" (or "chunks" of data) that occur frequently and it reserves codewords of more bits to encode less frequently occurring chunks in the data. The result is that the total file size is smaller than the uncompressed version. However, in order to read it, the file must be decompressed.

In bioinformatics, many of the files you deal with will be compressed, because that can save many terabytes of disk space. Most often, files will be compressed using the `gzip` utility, and they can be uncompressed with the `gunzip` command. Sometimes you might want to just look at the fist part of a compressed file. If the file is compressed with `gzip`, you can decompress to *stdout* by using `gzcat` and then pipe it to `head`, for example.

A central form of compression in bioinformatics is called `bgzip` compression which compresses files into a series of blocks of the same size, situated in such a way that it is possible to *index* the contents of the file so that certain parts of the file can be accessed without decompressing the whole thing. We will encounter indexed compressed files a lot when we start dealing with BAM and vcf.gz files.

2.8.2 Hashing

The final topic we will cover here is the topic of hashing, an in particular the idea of "fingerprinting" files on ones computer. This process is central to how the git version control system works, and it is well worth knowing about.

Any file on your computer can be thought of as a series of bits, 0's and 1's, as, fundamentally, that is what the file is. A *hashing algorithm* is an algorithm that maps a series of bits (or arbitrary length) to a short sequence of bits. The SHA1 hashing algorithm maps arbitrary sequences of bits to a sequence of 160 bits.

There are $2^{160} \approx 1.46 \times 10^{48}$ possible bit sequences of length 160. That is a vast number. If your hashing algorithm is well randomized, so that bit sequences are hashed into 160 bits in a roughly uniform distribution, then it is exceedingly

unlikely that any two bit sequences (i.e. files on your filesystem) will have the same hash ("fingerprint") unless they are perfectly identical. As hashing algorithms are often quite fast to compute, this provides an exceptionally good way to verify that two files are identical.

The SHA1 algorithm is implemented with the `shasum` command. In the following, as a demonstration, I store the recursive listing of my `git-repos` directory into a file and I hash it. Then I add just a single line ending to the end of the file, and hash that, to note that the two hashes are not at all similar even though the two files differ by only one character:

```
[~]--% ls -R Documents/git-repos/* > /tmp/gr-list.txt
[~]--% # how many lines is that?
[~]--% wc /tmp/gr-list.txt
   93096   88177 2310967 /tmp/gr-list.txt
[~]--% shasum /tmp/gr-list.txt
1396f2fec4eebdee079830e1eff9e3a64ba5588c  /tmp/gr-list.txt
[~]--% # now add a line ending to the end
[~]--% (cat /tmp/gr-list.txt; echo) > /tmp/gr-list2.txt
[~]--% # hash both and compare
[~]--% shasum /tmp/gr-list.txt /tmp/gr-list2.txt
1396f2fec4eebdee079830e1eff9e3a64ba5588c  /tmp/gr-list.txt
23bff8776ff86e5ebbe39e11cc2f5e31c286ae91  /tmp/gr-list2.txt
[~]--% # whoa! cool.
```

2.9 Unix: Quick Study Guide

This is just a table with quick topics/commands/words in it. You should understand each and be able to tell a friend a lot about each one. Cite it as 2.1

TABLE 2.1: Terms/ideas/etc. to know forward and backward

bash	absolute path	relative path
`/` at beginning of path	`/` between directories	home directory
`~`	current working directory	`pwd`
`cd`	`.`	`..`
`cd -`	basename	PS1
TAB-completion	`ls` (`-a`, `-d`, `-R`)	globbing
*	?	[0-9]
[a-z]	[^CDcd]	{png,jpg,pdf}

TABLE 2.1: Terms/ideas/etc. to know forward and backward

`echo`	`man` *command*	`mkdir`
`mv`	`cp`	`rm`
`cat`	`head`	`less`
stdout	*stdin*	*stderr*
`>`	`<`	`|`
`ln -s`	symbolic link	PATH
`-rw-r--r--`	.bashrc	.bash_profile
`sort`, `-n -b -k`	`paste`	`cut`, `-d`
`tar`, `-cvf, -xvf`	`gzip`	`du`, `-h -C`,
`wc`	`date`	`uniq`
cntrl-c	cntrl-r	`#`
up-arrow/down-arrow	`chmod`, `ug+x`, `664`	`grep`

3

Shell programming

In our first foray into Unix and the shell, we restricted ourselves mostly to navigating the file system, handling files, and working with streams of data (via redirection and pipes). These are all crucial skills, but the bash shell becomes truly powerful when we start to adopt it as a sort of programming language. That's right, even though the functionality of bash is geared toward running jobs and calling commands, it still exhibits most of the features expected in a programming language, like variables, iteration and flow control.

3.1 An example script

We start this chapter by taking a look at a short bash program (typically called a *script*) that the author wrote in order to efficiently download (clone, really) repositories from GitHub that have been submitted by students to GitHub Classroom. The program is not long, but exhibits many useful features of bash as a programming language. If you are reading this, not in the context of a class with lectures, just go ahead and read through it and see if you can figure out what is going on in each line of the script. Afterward, we will address many features of bash by referencing different parts of the script.

The script, which happens to be stored in a file called `clone-classroom-repos.sh` is printed below, with linenumbers, since we will be referring back to specific sections of the script later.

```
1  #!/bin/bash
2
3  # define a function to print the usage or "help" for the script
4  function usage {
5        echo Syntax:
6        echo "  $(basename $0)  GH_Prefix  Repo_Prefix  Branch  Dir
7
8        GH_Prefix: the URL of the GitHub site where the repository exists.
```

```
      Repo_Prefix: the prefix of the name of each repository to be cloned.
      Branch: the name of the branch to create and switch to in the repository,
         once the repo has been cloned.
      Dir: path to the directory (will be created if necessary) to clone all
         the repositories to.

   Example:

      $(basename $0)  https://github.com/CSU-con-gen-bioinformatics-2020  illumina-video-q
      "
      echo
}

#  test for right number of required args.  If not, print usage message
if [ $# -ne 4 ]; then
    usage;
    exit 1;
fi

# copy positional parameters into other variables
GHP=$1
RP=$2
BRANCH=$3
DD=$4

# assign string with student GitHub handles into a variable
GHNAMES="AmandaCicchino
BrennaF
CaitlinWells
EllenMCampbell
FayDong
LibbyGH
NathanPhipps
RGCheek
Ronan17
abeulke
carolazari
cbossu
ccolumbu
elenacorrea
eriqande
jenleon07
kimhoke
kruegg
```

```
53 lauracgoetz
54 mdrod110
55 mgdesaix
56 raven-wings
57 seamus100
58 taylorbobowski
59 wcfunk"
60
61 # assign my GitHub username to the variable USER
62 USER=eriqande
63
64 # assign the current working directory to the variable RUNDIR
65 RUNDIR=$PWD
66
67 # make a new directory named whatever the user wanted for the output directory
68 mkdir -p $DD
69
70 # make variables to hold log and error file names
71 LOG=${PWD}/${RP}log
72 ERR=$LOG.stderr
73
74 # print the date/time when the process is starting
75 echo "STARTING at $(date)"
76
77 # make a clean slate. remove any files with the name
78 # of the error output file
79 rm -f $ERR
80
81 # cycle over the student GitHub names, and for each one *do*
82 # the commands that appear before the *done* keyword. Indenting
83 # is used to make it easier to read, but is not essential.
84 for L in $GHNAMES; do
85
86     echo "Working on $L, starting at $(date)"  # print a progress line to stdout
87     REPO=$GHP/${RP}$L     # combine variables into new variables that
88     echo $REPO            # hold the URL for the repository to be
89     DEST=$DD/$L           # cloned and the path where it should be cloned to
90
91     # store the commands themselves into variables. Note the
92     # use of double quotes.
93     CLONE_IT="git clone ${REPO/github.com/$USER@github.com} $DEST"
94     BRANCH_IT="git checkout -B $BRANCH"
95     PUSH_IT="git push -u origin $BRANCH"
96
```

```
97
98      # now, run those commands, chained together by exit-status-AND
99      # operators (so it will stop if any one part fails), while
100     # all the while appending error statements to the Error file. Run it
101     # all within an "if" statement so you can deliver a report as to
102     # whether the whole shebang succeeded or failed.
103     if $CLONE_IT 2>> $ERR  && \
104         cd $DEST && \
105         $BRANCH_IT 2>> $ERR  && \
106         $PUSH_IT 2>> $ERR  && \
107         cd $RUNDIR   # at the very end make sure to return to the original working directo
108     then
109         echo "FULL SUCCESS $L"
110     else
111         echo "FAILURE SOMEWHERE WITHIN $L"
112         cd $RUNDIR  # get back to the working directory from which the original command wa
113                     # so we are ready to handle the next student repo.
114     fi
115
116 done  # signifies the end of the for loop we are cycling over
```

If my current working directory is where the script resides, I can run it like this:

```
% ./clone-classroom-repos.sh
```

And if I wanted to be fancy, I could put the script in a directory (like `~/bin` perhaps) that I have included in my `PATH` variable. In which case I could run it like:

```
% clone-classroom-repos.sh
```

from anywhere on my computer.

When I run the script in any of those two ways, because I have not provided the proper number of *arguments* to the command, it returns a message telling me what syntax is required to use it (i.e., its *usage syntax*):

```
% clone-classroom-repos.sh
Syntax:
  clone-classroom-repos.sh  GH_Prefix  Repo_Prefix  Branch  Dir

      GH_Prefix: the URL of the GitHub site where the repository exists.
```

```
      Repo_Prefix: the prefix of the name of each repository to be cloned.
      Branch: the name of the branch to create and switch to in the repository,
         once the repo has been cloned.
      Dir: path to the directory (will be created if necessary) to clone all
         the repositories to.

   Example:

      clone-classroom-repos.sh  https://github.com/CSU-con-gen-bioinformatics-2020  illumi
```

That is handy, and the code to do it exists in the script itself. Looking at the output, how many arguments do you think the script is expecting?

Now, if I wanted to clone all of the student GitHub repos associated with the `illumina-video-questions` homework set, and then, once cloned, set up a new git *branch* called `eric-edits` so that I can make edits and/or comments and send those to students via a pull request, here is the command I would give (remembering, again that the `%` signifies the command prompt, here):

```
% clone-classroom-repos.sh  https://github.com/CSU-con-gen-bioinformatics-2020  illumina-v
```

And when I do, I see output like this:

```
STARTING at Thu Feb 13 06:02:05 MST 2020
Working on AmandaCicchino, starting at Thu Feb 13 06:02:05 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-AmandaCicchino
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS AmandaCicchino
Working on BrennaF, starting at Thu Feb 13 06:02:07 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-BrennaF
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS BrennaF
Working on CaitlinWells, starting at Thu Feb 13 06:02:08 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-CaitlinWells
FAILURE SOMEWHERE WITHIN CaitlinWells
Working on EllenMCampbell, starting at Thu Feb 13 06:02:09 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-EllenMCampbell
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS EllenMCampbell
Working on FayDong, starting at Thu Feb 13 06:02:10 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-FayDong
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS FayDong
Working on LibbyGH, starting at Thu Feb 13 06:02:12 MST 2020
```

```
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-LibbyGH
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS LibbyGH
Working on NathanPhipps, starting at Thu Feb 13 06:02:14 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-NathanPhipps
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS NathanPhipps
Working on RGCheek, starting at Thu Feb 13 06:02:15 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-RGCheek
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS RGCheek
Working on Ronan17, starting at Thu Feb 13 06:02:17 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-Ronan17
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS Ronan17
Working on abeulke, starting at Thu Feb 13 06:02:19 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-abeulke
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS abeulke
Working on carolazari, starting at Thu Feb 13 06:02:21 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-carolazari
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS carolazari
Working on cbossu, starting at Thu Feb 13 06:02:23 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-cbossu
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS cbossu
Working on ccolumbu, starting at Thu Feb 13 06:02:25 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-ccolumbu
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS ccolumbu
Working on elenacorrea, starting at Thu Feb 13 06:02:26 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-elenacorrea
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS elenacorrea
Working on eriqande, starting at Thu Feb 13 06:02:28 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-eriqande
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS eriqande
Working on jenleon07, starting at Thu Feb 13 06:02:29 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-jenleon07
FAILURE SOMEWHERE WITHIN jenleon07
Working on kimhoke, starting at Thu Feb 13 06:02:30 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-kimhoke
FAILURE SOMEWHERE WITHIN kimhoke
```

```
Working on kruegg, starting at Thu Feb 13 06:02:30 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-kruegg
FAILURE SOMEWHERE WITHIN kruegg
Working on lauracgoetz, starting at Thu Feb 13 06:02:30 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-lauracgoetz
FAILURE SOMEWHERE WITHIN lauracgoetz
Working on mdrod110, starting at Thu Feb 13 06:02:31 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-mdrod110
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS mdrod110
Working on mgdesaix, starting at Thu Feb 13 06:02:32 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-mgdesaix
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS mgdesaix
Working on raven-wings, starting at Thu Feb 13 06:02:34 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-raven-wings
FAILURE SOMEWHERE WITHIN raven-wings
Working on seamus100, starting at Thu Feb 13 06:02:34 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-seamus100
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS seamus100
Working on taylorbobowski, starting at Thu Feb 13 06:02:36 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-taylorbobowski
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS taylorbobowski
Working on wcfunk, starting at Thu Feb 13 06:02:37 MST 2020
https://github.com/CSU-con-gen-bioinformatics-2020/illumina-video-questions-wcfunk
Branch erics-edits set up to track remote branch erics-edits from origin.
FULL SUCCESS wcfunk
```

From that report, it is clear that it takes about 1 to 2 seconds to handle each repository. If I were doing each repository by hand, (i.e. cloning and branching through a GUI interface like RStudio's) each repository would probably take me about 30 seconds to a minute, with a lot of copying and pasting and chance for errors, and I would have destroyed my wrists with all the repetitive tasks. So, this is a HUGE deal.

It is also easy to scan through the results and see, "Holy Moly! These are some dedicated students!" Everyone has successfully submitted their homework repositories (and thus we were `SUCCESS`-ful in cloning them), except for a handful who were traveling or otherwise occupied and had warned me they wouldn't be able to do the assignment.

When all is said and done, I have the following git repositories on my laptop which I can peruse at my leisure:

```
% ls /tmp/illumina-questions/
AmandaCicchino/ FayDong/        RGCheek/        carolazari/     elenacorrea/    mgdesaix/
BrennaF/        LibbyGH/        Ronan17/        cbossu/         eriqande/       seamus100/
EllenMCampbell/ NathanPhipps/   abeulke/        ccolumbu/       mdrod110/       taylorbobo
```

Additionally, in the directory where I ran the command, I have a file called `illumina-video-questions-log.stderr` that gives me a more detailed report of things when they worked or failed.

If you are new to Unix, then the above script likely appears a bit daunting. Our goal by the end of the chapter is to have described every little piece of bash syntax needed, so that you will be able to read and understand the above script. You will thus also be in a good position to start writing your own scripts to automate tasks and analyses on your computer.

We will start with an overview of the structure of a script and then delve into specific areas of syntax. For each area of syntax, we will provide some examples, and then leave some openings for you, the reader, to try your own hand at implementing each pattern that you see.

3.2 The Structure of a Bash Script

A bash script is merely a text file that is a collection of different command lines, one after the next, which the bash shell will run in sequence—one after the other. If you want to run the script, you must make sure you have set its permissions to include execute permissions (i.e., `chmod ug+x script.sh`).

It is important to point out however, that the bash programming syntax that we will be describing in this chapter is not solely useful in the context of scripts that are stored in files. Rather, all the programming syntax can still be used directly on the shell command line itself! This means you can employ all the little tricks you will learn in this chapter while directly "hacking away" at the command line. In this context, it is worth noting that if you want to write multiple distinct commands on a single line, *as if they were on separate lines*, you can separate them with a semicolon, `;`. For example:

```
echo "Put this in a file! (and catenate it later...twice!)" > tmp.txt
cat tmp.txt
cat tmp.txt
```

```
## Put this in a file! (and catenate it later...twice!)
```

```
## Put this in a file! (and catenate it later...twice!)
```

Is equivalent to:

```
echo "Put this in a file! (and catenate it later...twice!)" > tmp.txt; cat tmp.txt; cat tm
```

```
## Put this in a file! (and catenate it later...twice!)
## Put this in a file! (and catenate it later...twice!)
```

Sometimes when you are writing a script, (or even working on the command line) you might want a very long expression to be treated as being all part of the same command line, even though you would like to break it up over multiple lines. A backslash (\) immediately followed by a line ending (i.e., the "return" key) has the effect of treating the lines that it separates as all being on the same line.

```
# this:
clone-classroom-repos.sh  \
    https://github.com/CSU-con-gen-bioinformatics-2020  \
    illumina-video-questions- \
    erics-edits  \
    /tmp/illumina-questions

# is the same as this:
clone-classroom-repos.sh  https://github.com/CSU-con-gen-bioinformatics-2020  illumina-vid
```

Using backslashes (and indenting) in this way can sometimes dramatically improve readability of your scripts.

You might also notice a number of lines or statements in the script above that start with a `#`. The `#` is known as a *comment character*. When the bash interpreter is reading the script, it *ignores* the comment character *and anything following it* until the *end of the line*. This makes it quite convenient to pepper your scripts with notes to yourselves or others that can be *extremely* helpful when you come back to a piece of code and are trying to remember what it does! Example:

```
# this might report an error if there is no file
# named SillyRidiculous.what
ls -l SillyRidiculous.what
```

```
## ls: SillyRidiculous.what: No such file or directory
```

(Note: each line of the the *output* shown above follows two `#` symbols. This is not commenting. It is just the convention that the 'bookdown' package uses to

signify that what it is showing you is output, rather than input. No relation, really, to commenting…

In the `clone-classroom-repos.sh` script, listed above, on the very top line (line 1) you will see a special statement that follows a `#` comment character. This is one of the few cases you will see when the contents after a comment character are *not* ignored (the other place you will see this is when preparing additional statements for schedulers for high-performance computing systems!). In this case the combination `#!` on the first line is telling Unix to get ready to learn how to interpret the contents of the script:

```
#!/bin/bash
```

The part after the `#!` is just the path to the program `bash` that implements the bash shell. It is telling Unix how to run the script. While it is common practice to put this top line on a script, on most systems, if the line is absent, Unix will interpret the script using bash, anyway.

If you ever find yourself having a hard time remembering the order of those first two symbols (i.e., is it `#!` or `!#`?), just remember that it is sometimes called the *shebang*. You know the first part should be a comment character so the line is treated a little differently, but then that has to be followed by the "bang" which is `!`.

3.2.1 A bit more on ; and &

Recall the semicolon: it provides a way to combine multiple commands on a single line. In fact, when you think of putting multiple commands on different lines, you could think of each one being followed by a semicolon. Like this:

```
# this
echo one
echo two

# is the same as
echo one;
echo two;

# is the same as
echo one; echo two;
```

However, there is another character, beyond the semicolon, with which you might follow commands. It is the *single* `&` symbol. Thus, you could use it like a semicolon, writing and executing this:

```
echo one & echo two &
```

But, if you do that on your Unix system, you should see some numbers and a report about jobs, like this:

```
[1] 1007
[2] 1008
one
two
[1]-  Done                    echo one
[2]+  Done                    echo two
```

Whoa! What the heck is going on there? The `&` symbol means, "run the command that came before it, but don't bother waiting for it to be done before running the next one." As a consequence of this, when you run a command followed by a `&`, the computer returns to you the job ID numbers for the jobs that have been started, and when they are done, it also tells you that the jobs that have completed.

If you want to run multiple jobs on your own computer, the `&` syntax can be helpful. But, most of the time as a bioinformatician, you will be vying with countless others to run jobs on a large server or cluster. In those cases there are more refined systems for allocating jobs, and, as a consequence, you may rarely use the single `&` syntax when working on a high performace computing system.

3.3 Variables

When your goal is to script up repetitive tasks, one of the main ingredients to your success is bash's ability to *assign values* to *variables*, and then later, retrieve those values and replace a variable in your script with its value, a process known as *variable substitution*. Both *variable assignment* and *variable subtitution* happen all over the `clone-classroom-repos.sh` script, in lines like:

```
USER=eriqande
LOG=${PWD}/${RP}log
ERR=$LOG.stderr
```

We are going to break this down because it is of central importance.

3.3.1 Assigning values to variables

The syntax to assign a value to a variable is, "put the variable on the left, follow it with an equals sign **with absolutely no spaces around it**, and the put the value on the right," like this:

```
VAR=value
HUNGRY="My cat, Oliver"
SWEET="A chocolately treat"
MY_FILE=/Users/eriq/Documents/git-repos/eca-bioinf-handbook/eca-bioinf-handbook.Rproj
lowercase_variables_work_too="oh yeah."
Or_even_MIXtures_of_cases=boing
```

Identify the variables and the values in the above.

The names of variables must start with either an upper- or lowercase letter or an underscore. After that initial character, valid variable names can then include any combination of underscores, upper- and lowercase letters and numerals.

In the following, identify the variable names that are valid and those that are not on each line. After you have made your choices, paste all these lines into your terminal to see which ones work and which do not.

```
vaRiAble=value
1_tough_cookie="hard to eat with false teeth"
_bring_it_on="A fine musical"
PLATE.FIFTEEN=/home/me/labwork/plate_15
```

Ha! That is a pretty easy task because syntax highlighter in 'bookdown' colors variables differently than other parts of a script. Oh well, you still get the point!

The value of the variable, on the other hand can be pretty much any string (as long as it doesn't confuse the shell with characters like `&`, `;`, or `!`). The shell typically understands that strings are delimited by whitespace, so, if your string (value) should include multiple words separated by spaces, you must enclose them in quotation marks:

```
NAME="Eric C. Anderson"
```

You can use either pairs of single quotes (`'this is single-quoted'`) or double quotes (`"this is double-quoted"`), but the shell treats these very differently, as we will see. Most of the time, in bioinformatics, you will want to be using double quotes.

If you want to include, in a variable's value, characters that have special

meaning to the shell, like `*`, `[`, `{`, `&`, and `;`, among others, you must enclose the string inside quotation marks to assign it to your variable:

```
FOO="bar&grill"  # works
FOO=bar&grill    # won't work the way you want it to
```

3.3.2 Accessing values from variables

This is called "variable substitution." Remember, when you want something from someone, (or even just from a variable) it might cost you some money. Which is how you can remember that you need to use the `$` to access values from variables. The `$` tells the shell that you want the value of the variable, in a process called *variable substitution*. It is called that, because if you write `$VAR`, somewhere in a command line, then the shell will happily go along and substitute the *value* of the variable `$VAR` in place of the variable itself, *after* which the shell will evaluate the command line.

If you have a variable called `VAR`, the writing `$VAR` subtitutes its value on the command line. The same occurs if you write `${VAR}`. The latter is a little more formal, but is also, in some sense a little more flexible, because it lets you append text immediately after the variable:

```
FIRST_PART=oxy

# this works:
echo "I like the word ${FIRST_PART}moron"

# this doesn't
echo "I like the word $FIRST_PARTmoron"

# but this would
echo "Remove the stain with $FIRST_PART-clean"
```

```
## I like the word oxymoron
## I like the word
## Remove the stain with oxy-clean
```

In the case above that does not work, the shell is subtituting the value of the variable `FIRST_PARTmoron` which actually does not exist, so it substitutes nothing (and doesn't even give you an error). This type of mistake typically occurs when you forget the the underscore is part of a valid variable name, like:

```
GENUS=Oncorhynhus
SPECIES=mykiss

cd $GENUS_$SPECIES  # Fail!

cd ${GENUS}_${SPECIES}  # Works!
```

Now it is your turn: save your three favorite foods into the variables ONE, TWO, and THREE, and then use the `echo` command to print `My three favorite foods are...` where you include those foods via variable substitution:

Are we starting to feel the power of Unix scripting yet?

3.3.3 What does the shell do with the value substituted for a variable?

This is a great question, and it gets at the heart of why Unix is so powerful for scripting. Recall that variable substitution occurs before command evaluation. So, basically, after variable substitution, the shell has a command line that includes the values instead of the variables. Then it just evaluates that command line. So, what happens to the values that have been substituted for the variables depends on what context they appear in, in the command line!

Follow along with this example:

```
# Start by assigning to DIR the absolute path of a directory you often go to
# Note, you should use a path from your own computer!
DIR=/Users/eriq/Documents/git-repos/eca-bioinf-handbook

# Now, note that we can do different things with that variable
# depending on how/where we put it in a command line.

# print it
echo $DIR

# list its contents
ls $DIR

# from your home directory (reached with `cd` and nothing after it)
# you can go directly to DIR like so:
cd
```

```
cd $DIR

# If you just put $DIR on the command line by itself, the shell
# interprets it as a command, and tries to execute it, which give an error:
$DIR

# On the other hand, if you wanted to make a new variable:
GODIR="cd $DIR"

# then, that would work as a command on its own:
cd
$GODIR
```

Oh! Now I am getting excited. Notice that in the `clone-classroom-repos.sh` script I did this quite a bit, making a few different command lines…

```
    # store the commands themselves into variables. Note the
    # use of double quotes.
    CLONE_IT="git clone ${REPO/github.com/$USER@github.com} $DEST"
    BRANCH_IT="git checkout -B $BRANCH"
    PUSH_IT="git push -u origin $BRANCH"
```

…that I would then call later:

```
    $CLONE_IT 2>> $ERR  && \
        cd $DEST && \
        $BRANCH_IT 2>> $ERR  && \
        $PUSH_IT 2>> $ERR  && \
        cd $RUNDIR
```

While you can count on subtituted variables to be evaluated in context, the above-displayed behaviors of the shell don't always work. Because of the intricacies of how the bash shell command parser works, and the order of parsing and variable substition, there are times when the string which is a substituted variable *will not* be evaluated exactly as that same string would be evaluated if it were just text in a script. Typically the differences in behavior are found when `=` signs or redirects are found in the string.

An example should make this clear:

```
# here is a command that is a simple string
echo me oh my, i like pie.
```

```
# if we run it as a command we see:
me oh my, i like pie.

# here is that same command stored in a variable
boing="echo me oh my, i like pie."

# Variable substitution for boing on the command line looks like:
$boing

# and the result we see is:
me oh my, i like pie.

# However, if we type this:
echo me oh my, i like pie. > /tmp/out

# then the the file /tmp/out will hold:
me oh my, i like pie.

# But if we say
bonk="echo me oh my, i like pie. > /tmp/out"

# And do:
$bonk

# We get:
me oh my, i like pie. > /tmp/out

# which is clearly not the same thing
```

In the above, example, when > occurred within a variable, the command parser did not recognize it as the redirection operator.

Here is another example: imagine that we want to assign three values to three different variables. If we just assigned them on the command line we could do something like this:

```
# simple variable assignment
A=one; B=two; C=three

# after that, if we do:
echo $A $B $C

# we get
one two three
```

But, what if we wanted to assign three values to three variables within a string that gets assigned to a variable, and then substitute that variable on the command line to actually assign those variables. For example:

```
ASSIGNMENTS="D=four; E=five; F=six"

# then try to substitute ASSIGNMENTS to actually make
# the variable assignments:
$ASSIGNMENTS

# bash replies with an error message
-bash: D=four;: command not found
```

It is clear that bash is not able to interpret the `=` sign as the assignment operator in this context.

We can solve this problem by using the `eval` keyword to tell bash to explicitly evaluate the command line a second time after variable substitution has occurred. Observe. In the latter case:

```
# create variable that is a line with assignments
ASSIGNMENTS="D=four; E=five; F=six"

# evaluate that line after variable substitution:
eval $ASSIGNMENTS

# see that the variables D, E, and F have values assigned:
echo $D, $E, $F

# cool!
```

And in the former:

```
# assign complete command line with redirection to bonk
bonk="echo me oh my, i like pie. > /tmp/out"

# evalute command line after $bonk's value is substituted
eval $bonk

# now /tmp/out holds the contents:
me oh my, i like pie.
```

The `eval` keyword is not often used in `bash`, but it is very useful in bioinformatics for evaluating command lines, *that include their redirection specifiers*, and which have been stored in a variable. Why is this important? If your script stores a command line (after variable subtstitution) in a variable, then you can print that command line before evaluating it. When developing your scripts this makes it easy to test your command lines (just echo it, copy it, and paste it into a shell you have open for testing). When running scripts, if you print out each line to a log, it is easier to go back to ones that failed and figure out why.

Your Turn: Make a variable called `YELL_IT` that holds the command line that would print the line "Oh, I just gotta be me" and redirect it into the file `yawp.txt`. Once you have done that, do variable substitution without, and then with, the `eval` keyword. Which one actually gets the job done?

Remember that in RStudio, if you highlight some text and do CMD-Option-Return on a Mac, the text gets sent to the RStudio Unix Terminal. (I suspect the PC equivalent is cntrl-alt-return). That can be helpful for quickly testing lines you have written.

3.3.4 Double and Single Quotation Marks and Variable Substitution

Quite often you will want to save a value to a variable that, itself, includes other variables. In other words, you want to do variable subtition on the value you are assigning to the variable. Double quote let you do this: variable substition will proceed within double quotes:

```
FOO=sandwiches
BAR="I like $FOO"
echo $BAR
```

```
## I like sandwiches
```

However, inside single quotes, variable subtitution will not occur:

```
FOO=sandwiches
BAR='I like $FOO'
echo $BAR
```

```
## I like $FOO
```

Each behavior has its uses, but most of the time, as I said, you will be wanting to use double quotes.

Now it is your turn: Save three pairs of foods that you like (for example, "cookies & cream" or "bananas & walnuts") into three variables, `PAIR_ONE`, `PAIR_TWO`, and `PAIR_THREE`. Then combine those into a variable called `SENTENCE` and print it with `echo`, so that the result looks something like: "I like cookies & cream, and bananas & walnuts, and rooibos and milk."

3.3.5 One useful, fancy, variable-substitution method

Bash is full of fancy variable-substitution embellishments. One that I use all the time replaces strings in a variable with other strings. Check this out:

```
FILE=my_picture.jpg
echo $FILE
echo ${FILE/jpg/png}
echo ${FILE/my/your}
```

```
## my_picture.jpg
## my_picture.png
## your_picture.jpg
```

That turns out to be some powerful stuff.

3.3.6 Variable arrays

Most programming languages allow you to store things in arrays. In R they are called *vectors* (or, more generally, *lists*). The same is true in `bash`: you can store a number of values into a single variable, and then access each individual value one at at time. This can be quite useful sometimes.

The syntax for assigning values to array variable is like this:

```
ArrayVariable=(words or things 'or stuff' separated 'by whitespace')
```

In other words, wrap the items inside a pair of parentheses, with white-space separating them, and the shell will break those up into different, *numbered* parts of the array variable. It is important to note that single-quoted groups of words are treated as single values that will be assigned to an array element as a group. (Though the same is not true of double-quoted values…)

Once values are stored in an array, how do we access them, i.e., what sort of variable subtitution can we do?

Well, if we just do the traditional variable substitution, like `${ArrayVariable}` or `$ArrayVariable`, then we just get the first element:

```
# doing this
echo $ArrayVariable

# produces this:
words
```

That is not super helpful. So, to get each separate element we can *subscript* the array by adding a number in square brackets inside the curly braces, like this:

```
# Doing this:
echo ${ArrayVariable[2]}

# gives:
things
```

The subscripting is done with 0 as the starting value, so `${ArrayVariable[0]}` will give `words`, and `${ArrayVariable[1]}` gives `or`. Here is a `for` loop (see below) that prints each of the different subscripts followed by their associated values:

```
# this loop...
for i in {0..5}; do
  echo $i: ${ArrayVariable[$i]}
done

# produces output like:
0: words
1: or
2: things
3: or stuff
4: separated
5: by whitespace
```

There are a few more important ways of accessing array variable values:

You can get all the values at once as a single string. There are two ways you can do this: either `${ArrayVariable[*]}` or `${ArrayVariable[@]}`. These

two methods differ only when you have wrapped the variable up in double quotation marks, but that involves some serious bash arcana.

```
# when just printing output, both of these give the same result
echo ${ArrayVariable[*]}
echo ${ArrayVariable[@]}

# producing:
words or things or stuff separated by whitespace
words or things or stuff separated by whitespace
```

And finally, the *length* of the array (meaning, how many elements are in it) is found with a peculiar syntax `${#ArrayVariable[@]}`

```
# this:
echo ${#ArrayVariable[@]}

# produces:
6
```

Although bash arrays can be quite useful, I must admit that I only use them in a few bioinformatic situtations—primarily when I want to break a single row of white-space delimited columns (that I grabbed from a file, for example) into its consituent words that I can then manipulate.

Your turn: Create a variable called `MY_ARRAY` in which each element is the first name of 5 different people in our bioinformatics class. Then print the 1st, 2nd and 5th of them each on a separate line.

3.4 Evaluate a command and substitute the result on the command line

Sometimes what you want to put on a command line isn't just a variable you have previously defined but rather the result of a command that is executed. We see this in `clone-classroom-repos.sh` around line 75:

```
echo "STARTING at $(date)"
```

In general, if you put a command inside a `$()`, like `$(command)` it means take the output of the command and insert it into the command line.

You can even assign the result to a variable, like `RESULT=$(command)`.

For example, try this:

```
# what do you think this is doing?
HOME_LIST=$(ls -l ~)

# what about this?
echo $HOME_LIST
```

Whoa! What happened to all my carriage returns? The results gets put onto the command line and parsed there. The command line parser sometimes treats line endings as just more whitespace, and it converts all runs of whitespace to a single space…

Your Turn: Make a three line script. In the first one, assign the output of the `date` command to the variable `BEFORE`, then give this command `sleep 5`, then in the third line, assign the output of the `date` command to the variable `AFTER`. Evaluate all three lines at once (by CMD-OPTION-Return or copying and pasting into the terminal.) Then, look at `BEFORE` and `AFTER` to compare.

3.5 Grouping/Collecting output from multiple commands: `(commands)` and `{ commands; }`

Quite often you may wish to think of the result of a group of commands (taken together such a group is called "list" in bash parlance) as being the result of a single command. Typical use cases are when you want to redirect the output from three separate commands into a single file. For example if you want to put the contents of two files, FileA and FileB, into a file called "Both," but separate the contents of FileA and FileB by a short line of x's, you could do this:

```
cat FileA > Both
echo xxxxxxxxxxx >> Both
cat FileB >> Both
```

But, it is easier to see what is going on and to maintain code that looks like this:

```
(cat FileA; echo xxxxxxxxxxx; cat FileB) > Both
```

or like:

```
{ cat FileA; echo xxxxxxxxxxx; cat FileB;} > Both
```

Both parentheses and curly braces can be used to group commands into one *grouped* command, thereby making it easy to redirect the output from that command. These two forms of grouping have subtle differences.

When you group commands into parentheses, all the commands get evaluated in a separate *subshell*. By contrast, grouped commands inside curly braces are all evaluated within the current shell. In many cases this will make almost no difference to you. However, if you are assigning variable values within the grouped commands, then, using parentheses for grouping, you won't have access to those variable values in the current shell:

```
# include a variable assignment in parentheses
(cat FileA; echo xxxxx; cat FileB; NewVar=15) > Both

# if you try this:
$NewVar

# the shell knows nothing about it
```

Using curly braces, the variable assignments will be known in the current shell you are in:

```
# include a variable assignment in curly braces
{ cat FileA; echo xxxxx; cat FileB; NewVar=15; } > Both

# Now, if you try this:
$NewVar

# the shell knows that $NewVar has a value:
15
```

The syntax for using curly braces for grouping, however, is more finicky than it is with parentheses: the left curly brace cannot be touching anything on the right, and the last command in the group must be followed by a semicolon or a newline. Thus, both of these would fail:

```
{cat FileA; echo xxxxxxxxxxx; cat FileB;} > Both
{ cat FileA; echo xxxxxxxxxxx; cat FileB} > Both
```

3.6 Exit Status

When you run a command in Unix, that command might do a lot of different things, like print something to *stdout*, or copy a file around, or delete a file, or index a whole genome. Regardless of all the things a command might do, it also should let the operating system know whether it was successful or not. The *exit status* of a Unix command records whether it finished its task successfully or not. It is important to understand how exit statuses work so that you can design bioinformatic pipelines that will stop when something has gone wrong, and will let you know about that.

If a command exits normally (SUCCESS!) the exit status is 0. If the command does not exit normally (it may have been unsuccessful) then its exit status will be anything but 0. Some programs, when they fail, return a non-zero integer as their exit status, and the value can tell you what kind of error occurred. Other programs might just return 1.

Exit statuses are not typically seen by the user, but they do get passed to the shell. You can always access the exit status of the last command with `$?`. You can remember that because, with the question mark, it is kind of like you are asking the operating system, "What's up!!??".

Here is an example: if we try to list a file, using `ls`, that exists, the file gets listed and the exit status is 0 (SUCCESS!). If the file does not exist we get an error message and an exit status of 1 (NO_SUCCESS!).

```
# list a file we know exists
% ls -d ~/Documents
/Users/eriq/Documents/

# check exit status of last command
% echo $?
0

# list a file we are pretty sure does not exist
% ls I-doubt-this-file-exists.yeah.sure
ls: I-doubt-this-file-exists.yeah.sure: No such file or directory
```

```
# check the exit status of last command
% echo $?
1
```

While we don't get to "see" the exit status of a command without looking at `$?`, to the bash shell, when a command has completed on the command line, it effectively becomes the value of its exit status. This is important to understand when dealing with combinations of exit statuses.

3.6.1 Combinations of exit statuses

In bioinformatics, exit statuses can be helpful in a script to "decide" whether to continue processing the next command, based on whether the previous one failed or not. `bash` has an elegant way of implementing this in terms of the binary operators `&&` and `||` that *combine* exit statuses. The `&&` is a logical-AND combinining operator. If ES1 and ES2 are two exit statuses, then `ES1 && ES2` is "SUCCESS!" only if *both* ES1 and ES2 have exit statuses of "SUCCESS!". Here is a quick table:

ES1	ES2	ES1 && ES2
NO_SUCCESS	NO_SUCCESS	NO_SUCCESS
NO_SUCCESS	SUCCESS!	NO_SUCCESS
SUCCESS!	NO_SUCCESS	NO_SUCCESS
SUCCESS!	SUCCESS!	SUCCESS!

If you study this table for a moment, you will see that in the first two cases, the value of the combination, `ES1 && ES2` is apparent, just from knowing `ES1`. It doesn't matter whether `ES2` has an exit status of `SUCCESS!` or `NO_SUCCESS`; either way, because `ES1` has failed, we know that `ES1 && ES2` will be `NO_SUCCESS`.

This knowledge, combined with an understanding that the bash shell is typically very busy, and so is not going to do any extra work *that it does not need to do* will help you to understand the behavior of the shell when two (or more) commands are joined into a "compound command" with the `&&` symbol. Consider this:

```
command1 && command2
```

When bash looks at that, it sees that whoever wrote it wants to know the `&&`-combination of exit statuses of `command1` and `command2`. `bash` keeps that

thought in the back of its mind, and then starts working through the commands from left to right. If `command1` fails, the shell says, "Hey! At this point, I know the exit status of `command1 && command2`, so I am not even going to evaluate `command2`! On the other hand, if the exit status of `command1` is"SUCCESS¡', then the shell will proceed to evaluating `command2`, because it knows that if the exit status of `command1` was"SUCCESS!", then it must evaluate `command2` so as to get its exit status to properly evaluate `command1 && command2`.

The upshot of this is a construction like the following:

```
command1 && \
  command2 && \
  command3 && \
  command4
```

can be very useful in bioinformatics. This construction says, evaluate each command, if all the preceding commands were successful. If any of the commands fails, then none of the commands after them are evaluated. This is helpful if future steps depend on the successful completion of previous steps. In our example script, `clone-classroom-repos.sh` at the beginning of this chapter, we see the `&&` used on lines 103–106. This is saying, "if we didn't successfully clone the repository, then don't try to make a new branch in it, and if we didn't successfully make a new branch in it, then don't try to push that branch back to GitHub.

The opposite of `&&` is the `||` which combines exit statuses in an OR fashion with a table like the following:

ES1	ES2	ES1 || ES2
NO_SUCCESS	NO_SUCCESS	NO_SUCCESS
NO_SUCCESS	SUCCESS!	SUCCESS!
SUCCESS!	NO_SUCCESS	SUCCESS!
SUCCESS!	SUCCESS!	SUCCESS!

In this case, if `ES1` is `SUCCESS!` then we know that `ES1 || ES2` will have a combined exit status of "SUCCESS!" Consequently, we use the `||` to force evaluation of another command in case the previous one failed. Like this:

```
ls I-doubt-this-file-exists.yeah.sure || \
  echo "Aw shucks! That file aint there" > /dev/stderr
```

Note, in the above, we redirect the text, "Aw shucks! That file aint there" to

a file named `/dev/stderr`. That file, `/dev/stderr` is a special file: anything that you send into it gets immediately printed on *stderr*.

We end by noting that the numbers assigned to `SUCCESS!` and `NO_SUCCESS` do not accord with the 0's and 1's used in a standard "truth-table" context. We just have to deal with that. I find it much easier to think in terms of exit statuses of `SUCCESS!` and `NO_SUCCESS`, than in terms of the 0's and 1's, respectively, by which those statuses are represented in the computer.

3.7 Loops and repetition

By this point, you have probably heard, or been told, many times over, that Unix shell scripting is particularly good for taking care of repetitive tasks. But, by this point in this book, it might not yet be clear how that is the case. Wait no more! This section will reveal a wonderful construct called the `for` loop that lets you do a task repeatedly, each time setting the value of a variable to something different that you want to be applying some commands to.

The basic syntax of the `for` loop is:

```
# here the cycled variable is "i"
for i in some things separated by whitespace; do
  commands involving $i
done
```

For example:

```
for i in oranges bananas apples; do
  echo "I like $i"
done

# produces this:
I like oranges
I like bananas
I like apples
```

When you write a for loop in a script, it is good practice to indent the command lines that will get evaluated multiple times. This is particularly useful if you have nested for loops (one inside another) such as:

```
for fruit in pears figs; do
  for who in Mark Alice; do
    echo "$who likes $fruit"
  done
done

# which produces:
Mark likes pears
Alice likes pears
Mark likes figs
Alice likes figs
```

However, in bash (unlike Python, which is particularly obsessed with indentation) you are not required to indent things. In fact the above could have been written all on one line:

```
for fruit in pears figs; do for who in Mark Alice; do echo "$who likes $fruit"; done; done
```

It is also worth pointing out that, while many languages use curly braces to denote blocks of repeated code, `bash` uses the pair `do...done`, which I find to be quite cute.

As found with the variable `GHNAMES` in our example script, `clone-classroom-repos.sh`, you can also use variable subsitution to provide a list of terms to cycle over. Here is another small example:

```
ITEMS="cats dogs mice shrews"
for critters in $ITEMS; do
  echo $critters are vertebrates
done
```

You can also use globbing (path expansion) to provide a list of things (files, specifically) to cycle over. The following prints the path and the first two lines of all files in the directory `table_inputs` in this book's repository, at this point in the book's formation:

```
for i in table_inputs/*; do
  echo "======= file: $i  ========="
  head -n 2 $i
done

# this produces:
======= file: table_inputs/minimal-tmux.txt  =========
```

```
Within tmux?   ;  Command         ;  Effect
N              ; `tmux ls`          ; List any tmux sessions the server knows about
======= file: table_inputs/sam-columns-table.txt  =========
Column   &  Field   & Data Type  & Description
1        & QNAME    & String     & Name/ID of the read (from FASTQ file)
======= file: table_inputs/sam-flag-table.txt  =========
bit-#    &   bit-gram   &  $2^x$   &   dec    &   hex     & Meaning
1    &  $\bitsa$    &  $2^0$   &    1     &   0x1    & the read is paired (i.e. comes from
======= file: table_inputs/tmux-pane-strokes.txt  =========
Within tmux?   ;  Command         ;  Effect
Y              ; `<cntrl>-b /`       ; Split current window/pane vertically into two panes
```

Those files are the ones used as input to a few of the tables in the book.

If it turns out that you want to cycle over some integers, in order (whether increasing or decreasing), from a starting value to a stopping value, you can use curly braces and two dots, like this: `{1..5}`, like this:

```
for i in {1..5}; do echo The number is: $i; done

# this makes:
Number 1
Number 2
Number 3
Number 4
Number 5
```

And you can have negative numbers and a reverse order, too:

```
for i in {4..-2}; do echo The number is: $i; done

# this makes
The number is: 4
The number is: 3
The number is: 2
The number is: 1
The number is: 0
The number is: -1
The number is: -2
```

One cool thing to realize is that any output that goes to *stdout* from within a `for` loop—if it is not redirected from within the loop—effectively comes "flowing out" to *stdout* from right after the `done` keyword. So, you can redirect it in one swell foop from that point in your code like this:

```
for i in table_inputs/*; do
  echo "======= file: $i  ========="
  head -n 2 $i
done > file-to-redirect-it-all-into.txt
```

…or, you could even pipe it to another command, like this, to print just the first column of text of each line:

```
for i in table_inputs/*; do
  echo "======= file: $i  ========="
  head -n 2 $i
done | awk '{print $1}'
```

3.8 More Conditional Evaluation: `if`, `then`, `else`, and friends

We have already seen how exit statuses can be combined with `&&` or `||` to control the flow of a script (i.e., if this failed, don't do the next line…). There is also a traditional `if/then/else` construct in bash to control the execution of script on the basis of exit status. Thus, we can do something like:

```
if ls README.md; then
  echo "We found the README"
else
  echo "Can't find the README"
fi
```

Note that this opens an `if` block of code with `if` and then it closes it (after the `then` and the `else`) with a backward `if`: `fi`.

The general syntax is:

```
if exit_status; then
  Do this if exit_status = SUCCESS!
else
  Do this if exit_status = NO_SUCCESS
fi
```

There is also an else-if construct, that is named `elif`, with syntax like this:

```
if exit_status1; then
  Do this if exit_status1 = SUCCESS!
elif exit_status2; then
  Do this if exit_status1 = NO_SUCCESS and exit_status2 = SUCCESS!
else
  Do this if both were NO_SUCCESS
fi
```

Many of the times when you want to use an `if` construct, you will want to be *testing* things about files, or strings or variables, rather than assessing exit status of functions. Alas, all bash is capable of is assessing exit statuses. But hark! All is not lost becuase there is a function called `test` that let's you test if statements are true, and if they are, then it returns an exit status of 0 (SUCCESS!).

For example, to test if a file exists, you can use:

```
# this file does exist in the current directory on my system...
test -f eca-bioinf-handbook.Rproj

# do this to see what the exit status was
echo $?
```

Or, if we want to test if the value of a variable is the same as some string.

```
VAR=big_and_bad # set a variable to some string value

test $VAR = small_and_sweet

# get exit status, which will be NO_SUCCESS (1)
echo $?
```

The value of integer variables can be tested too. See `man test` for all the details.

Now, the thing to remember about the test function is that there is an intuitive-looking shorthand for writing it: that is to write its arguments between a `[` and a `]`, but not touching either of them.

So:

```
# this
test $VAR = small_and_sweet
```

```
# is equivalent to:
[ $VAR = "small_and_sweet" ];
```

Note that the RStudio bash shell seems to be doing something weird, I don't think the "last command" is quite what we think it should be, so `$?` does not seem to be reliable.

3.9 Finally...positional parameters

Way back when we started this chapter, in the example script, `clone-classroom-repos.sh`, right at the top we see lines like:

```
GHP=$1
```

What the heck is that `$1`. That is not proper syntax for a variable name! A variable name can't start with a number, after all. Aha! `$1`, `$2`, `$3`, and so forth are variables that store the *positional parameters* of a script. In other words if you write a script called `my_script.sh` and you invoke it with some words after it, like:

```
my_script.sh  BigFile.txt  small_file.txt  Yee-ha   'Oh Yeah'
```

Then, when that script is executing the code inside it, the value of `$1` will be `BigFile.txt`, the value of `$2` will be `small-file.txt`, the value of `$3` will be `Yee-ha`, and the value of `$4` will be `Oh Yeah`.

The variable `$#` inside the script holds the number of positional parameters, and, in our example script `[ $# -ne 4 ]` evaluates to `NO_SUCCESS` if `$#` is not equal to 4, in which case the script prints a message about how to use it.

Your Turn: Write a script in a separate file called "three-things.sh" that is expecting to take three positional parameters. In side the script, have it print the actual values passed to it in reverse order for example:

```
Third parameters is: -----
Second parameter is: -----
First parameters is: -----
```

Where `-----` would be replaced by the actual value of the positional parameters.

3.10 basename and dirname two useful little utilities

Paths to files and directories expand and are listed as the paths relative to the current working directory. Sometimes, you just want the name of the file. This is what basename gives you. For example:

```
# expand the filenames a couple directories down:
files="figure-creation/1.01-unix/*"

# print all those file names
echo $files

# that produces this output:
figure-creation/1.01-unix/file-hierarchy.dot figure-creation/1.01-unix/file-hierarchy.pdf

# do this to get just the filenames:
basename $files

# which produces this output:
file-hierarchy.dot
file-hierarchy.pdf
file-hierarchy.png
file-hierarchy.sh

# if you want just the relative path to the directories those files
# are in you can use dirname, but have to operate on one file at a time:
for i in $files; do
  dirname $i
done

# makes this output:
figure-creation/1.01-unix
figure-creation/1.01-unix
figure-creation/1.01-unix
figure-creation/1.01-unix
```

3.11 bash functions

In our example script, `clone-classroom-repos.sh`, we define a bash *function* called `usage` that prints a helpful message showing the syntax and an example invocation of the script. This provides an example of how you can write functions in `bash`. In all honesty, I only rarely use functions in `bash`, but it is good to know about nonetheless.

In `bash`, a function is just a collection of commands, grouped using curly braces, that will be evaluated when the function's name is issued on the command line or within a script. Not only will those lines be evaluated, but you can also pass *positional parameters* to the function by following its name with other words/tokens/values. The *positional parameters* within a function are distinct from the positional parameters within, say, the main script.

Functions can be defined, most transparently, by using the `function` keyword. As an example, here is a silly function, called `Silly` that takes two positional parameters and then merely prints them separated by monkey noises.

```
function Silly {
  echo "$1  Ooooh-oooh  Aaaah-aaah  $2"
}

# now, try
Silly foo bar
```

Another syntax you might see is to not use the *function* keyword, but rather follow the function's name with `()`:

```
Silly2() {
  echo "$1  Ooooh-oooh  Aaaah-aaah  $2"
}

# now, try
Silly2 boing bonk
```

You can even put it all on one line, but you have to make sure that you respect the curly-braces sensitivity to spacing and explicit line ending semicolons:

```
function Silly3 { echo "$1  Ooooh-oooh  Aaaah-aaah  $2";}
```

```
# now, try
Silly3 bing bap
```

3.12 reading files line by line

This is handy. Note the line can be broken into a shell array:

```
# this is an example of reading a file in which each row is delimited
# by whitespace, the second column is a file name and the the
# third column is a number
cat a_file | while read -r line; do
  A=($line);
  file=${A[1]};
  num=${A[2]};
done
```

3.13 Further reading

An excellent chapter on the development of Unix (Raymond, 2003)

A nice set of bash scripting tutorials can be found at https://ryanstutorials.net/bash-scripting-tutorial/

4

Sed, awk, and regular expressions

In the course of doing bioinformatics, you will be dealing with myriad different *text* files. As we noted in the previous chapters, Unix, with its file I/O model, piping capabilities, and numerous utilities, is well-suited to handling large text files. Two utilities found on every Unix installation—`awk` and `sed`—merit special attention in this context. `awk` is a lightweight scripting language that lets you write succinct programs to operate line-by-by line on the contents of text files. It is particularly useful for handling text files that have columns of data separated by white spaces or tabs. `sed` on the other hand, is particularly useful for automating "find-and-replace" operations on text files. Each of them is optimized to handle large files without storing a lot of information in memory, so they can be useful for quick operations on large bioinformatic data sets. Neither is a fully-featured programming language that you would want to write large, complex programs in (that said, I did once implement a complete program for full-sibling inference from multiallelic markers in `awk`); however they do share many of the useful text-manipulation capabilities of such languages, such as Perl and Python. Additionally, `awk` and `sed` are deployed in a consistent fashion across most Unix operating systems, and they don't require much time to learn to use effectively for common text-processing tasks. As a consequence `awk` and `sed` are a useful addition to the bioinformatician's toolbox.

Both `awk` and `sed` rely heavily on *regular expressions* to describe *patterns* in text upon which some operation should be performed. You can think of regular expressions as providing a succinct language for performing very advanced "find" and "find-and-replace" operations in a text file.

In this chapter we will only scratch the surface of what can be done with `awk` and `sed`. Indeed, there is an entire 432-page book[1] published decades ago by O'Reilly about `awk` and `sed`. Our goal here is to provide an introduction to a few basic maneuvers with both `awk` and `sed` and to describe instances where they can be useful, as well as to give an introduction to many (but certainly not all) of the patterns that can be expressed using regular expressions. We will start with a basic overview of how `awk` works. Then we will have a short look at regular expressions, then we will use those further in `awk`, and finally we will play with `sed` a little bit.

[1] http://shop.oreilly.com/product/9781565922259.do

In order to have a set of files to use in the examples, I have made a small GitHub repository called `awk-and-sed-inputs`. You can get that with

```
git clone https://github.com/eriqande/awk-and-sed-inputs
```

All of the examples in this chapter that use such external files assume that the current working directory is the repository directory `awk-and-sed-inputs`.

4.1 awk

4.1.1 Line-cycling, tests and actions

`awk` operates by cycling through a file (or an input stream from *stdin*) line-by-line. At each new line `awk` tests whether it should do anything with the contents of the line. If the answer to that test is "yes", then it executes the code that describes what should be done to the contents of the line. The scope of ways that `awk` can operate on text is quite wide, but, the most common use of `awk` is to print parts of the line, or do small calculations on parts of the line. The instructions that describe the tests and the actions are included in the `awk` "script", though, in practice, this script is often given to `awk` not as a file, but, rather, as text between single-quotes on the command line. The basic syntax looks like this:

```
% awk '
  test1  {action1}
  test2  {action2}
' file
```

Where `file` is the path of the file you want `awk` to read, and the "tests" and "actions" above are just placeholders showing where those parts of the script are written. The layout makes it clear that if `test1` is TRUE, `action1` will be executed, and if `test2` is TRUE, then `action2` is executed. Code describing the actions must always appear within a set of curly braces. We will refer to the text within those curly braces as an *action block*.

Two things to note: first, different tests and actions do not have to be written on separate lines. That makes things easier to read, but if you are writing these short scripts on the command line, it is often easier to put everything on a single line, like `'test1 {action1} test2 {action2}`, and that is fine. Second, if you don't supply a file path to `awk` it expects (and gladly processes) data from *stdin*.

This makes it easy to pipe data into `awk`. For example, putting the above two points together, the above `awk` script skeleton could have been written this way:

```
% cat file | awk 'test1  {action1} test2  {action2}'
```

Before we get too abstract talking about tests and actions, let's look at a few examples:

```
# print lines starting with @SQ in the SAM header of a file
awk '/^@SQ/ {print}' data/DPCh_plate1_F12_S72.sam

# print only those @SQ lines with a sequence name tag starting with "NC_"
awk '/^@SQ/ && /SN:NC_/ {print}' data/DPCh_plate1_F12_S72.sam

# same as above, but quit processing the file as soon as you hit
# an @SQ line with a sequence name starting with "NW_"
awk '
  /^@SQ/ && /SN:NC_/ {print}
  /^@SQ/ && /SN:NW_/ {exit}
' data/DPCh_plate1_F12_S72.sam


# print only lines 101, and 103 from the fastq file
gzcat data/DPCh_plate1_F12_S72.R1.fq.gz | awk 'NR==101 || NR==102 {print}'
```

The above examples show only two different actions (`print` and `exit`) and a variety of tests based on matching substrings in each line, or on which line number (`NR`, which stands for "number of the record").

Take a moment to make sure you understand which parts are the tests, and which are the actions in the above examples.

4.1.2 Column splitting, fields, `-F`, `$`, `NF`, `print`, `OFS` and `BEGIN`

Every time `awk` processes a line of text it breaks it into different *fields*, which you can think of as columns (as in a spreadsheet). By default, any number of whitespace (space or TAB) characters constitutes a break between columns. If you want field-splitting to be done using more specific characters, you can specify that on the command line with the the `-F` option. For example:

```
awk -F"\t" 'test {action}' file  # split lines into fields on single TAB characters
awk -F"," 'test {action}' file  # split lines into fields on single commas
```

```
awk -F":" 'test {action}' file  # split lines into fields on single colons
awk -F";" 'test {action}' file  # split lines into fields on single semicolons
# and so forth...
```

While in all those examples, the field separator is a single character, in full-blown practice, the field separator can be specified as a *regular expression* (see below).

Within an action block, you can access the different fields that `awk` has split out of the line using a `$` followed immediately by a field number, i.e., `$1`, `$2`, `$3`, … . When you get to fields with indexes greater than 9, you have to wrap the number in parentheses, like `$(10)`, `$(11)`,… . The special value `$0` denotes the whole line, not just one of its fields. (Note that if you give the action `print` without any arguments, that also just prints the whole line.) The special variable `NF` is the "Number of Fields", so `$NF` is the "last column."

Let's say we have a tab-separated file in `data/wgs-chinook-samples.tsv` that looks like this:

```
vcf_name    ID_Berk NMFS_DNA_ID BOX_ID  BOX_POSITION    Population  Concentration (ng/ul)
DPCh_plate1_A01_S1  CH_Plate1_1A    T144767 T1512   1A  Salmon River Fall   23.4    Fall
DPCh_plate1_A02_S2  CH_Plate1_2A    T144804 T1512   5F  Salmon River Fall   67.4    Fall
DPCh_plate1_A03_S3  CH_Plate1_3A    T145109 T1515   7G  Salmon River Spring 3.52    Spring
DPCh_plate1_A04_S4  CH_Plate1_4A    T145118 T1515   8H  Salmon River Spring 10.3    Spring
DPCh_plate1_A05_S5  CH_Plate1_5A    T144863 T1513   1A  Feather River Hatchery Fall 220 Fa
```

Then, if we wanted to print the first and the last columns (fields) we could do

```
awk -F"\t" '{print $1, $NF}' data/wgs-chinook-samples.tsv

# the first few lines of output look like:
vcf_name run_type
DPCh_plate1_A01_S1 Fall
DPCh_plate1_A02_S2 Fall
DPCh_plate1_A03_S3 Spring
DPCh_plate1_A04_S4 Spring
```

Notice that if there is no `test` before the action block, then the action is done on every line.

The `print` command prints variables to *stdout*, if you separate those variables with a comma, then in the output, they will be separated by the *output field sepator* which, by default, is a single space. You can set the *output field separator* using the `OFS` variable, in an action block that is forced to run at the beginning of execution using the special `BEGIN` test keyword:

```
awk -F"\t" '
  BEGIN {OFS=";"}
  {print $1, $NF}
' data/wgs-chinook-samples.tsv

# makes output like:
vcf_name;run_type
DPCh_plate1_A01_S1;Fall
DPCh_plate1_A02_S2;Fall
DPCh_plate1_A03_S3;Spring
DPCh_plate1_A04_S4;Spring
DPCh_plate1_A05_S5;Fall
```

The comma between the arguments to `print` is what makes `awk` print the output field separator between the items. If you separate arguments to `print` with just a space (and not a comma), there will be nothing printed between the two arguments on output. This, coupled with the fact that `print` will happily print any strings and numbers (in addition to variables!) you pass to it, provides a way to do some light formatting of the output text:

```
awk -F"\t" 'NR > 1 {print "sample_name:" $1, "run_type:" $NF}' data/wgs-chinook-samples.ts

# gives output like:
sample_name:DPCh_plate1_A01_S1 run_type:Fall
sample_name:DPCh_plate1_A02_S2 run_type:Fall
sample_name:DPCh_plate1_A03_S3 run_type:Spring
sample_name:DPCh_plate1_A04_S4 run_type:Spring
sample_name:DPCh_plate1_A05_S5 run_type:Fall
```

Note that if you don't provide an action block after a test, the default action, if the test is true, is assumed to be "print the whole line as is." Thus you can use `awk` to print matching lines simply like this:

```
awk '/regex/'
```

where `/regex/` just means "some regular expression," as is explained in the next section.

Now it's your turn!

```
# 1. Using the file data/wgs-chinook-samples.csv, print out the
# NMFS_DNA_ID the BOX_ID and the BOX_POSITION, separated by periods
```

4.1.3 A brief introduction to regular expressions

In a few of the above examples you will see tests that look like: `/^@SQ/`. This is a regular expression. In `awk`, regular expressions are enclosed in forward slashes, so the actual regular expression part in the above is `^@SQ`, the enclosing forward slashes are just delimiters that are telling `awk`, "Hey! The stuff inside here should be interpreted as a regular expression."

At this stage, you can think of a regular expression as a "search-string" that `awk` will try to match against the text in a line. At its simplest, a regular expression just describes how characters should match between the regular expression and the line being matched. For example:

```
/ACGGTC/
```

Is saying, "search for the word `ACGGTC`," which is something that might find in a DNA string.

If all that regular expressions did was express a search word (like your familiar find function in Microslop Word, for example), then they would be very easy to learn, but also very limited in utility. The good news is that all your standard numerals and upper- and lowercase letters work in regular expressions just like they do in your vanilla "find" function. So, all of the following regular expressions are just requesting that the word enclosed in the `/`'s be found:

```
/Fall/
/A01/
/plate/
/LN/
```

Regular expressions get more complicated (and useful) because some of the familiar punctuation marks have special meaning within a regular expression. These are called *metacharacters*. The fundamental metacharacters in `awk` are listed in Table 4.1. It is worth getting familiar with all of these as most are common to all languages that use regular expressions, such as R, python, and perl, so learning these will be helpful not just in using `awk`, but also in your programming, in general.

TABLE 4.1: Basic metacharacters used in regular expressions with `awk`.

Metacharacter(s)	Description
`.`	Match any single character
`[ ]`	Match any single character from a *character class* (see below)
`^`	Signifies/matches the *beginning* of the line or word (remember `/^@SQ/`)

Metacharacter(s)	Description
`$`	Signifies/matches the *end* of the line of word
`*`	Match zero or more occurrences of the character (or grouped pattern) immediately to the left
`?`	Match zero or one occurrences of the character (or grouped pattern) immediately to the left
`+`	Match one or more occurrences of the character (or grouped pattern) immediately to the left
`{n}`	Match n occurrences of the character (or grouped pattern) immediately to the left
`{m,n}`	Match any number between m and n occurrences of character (or grouped pattern) immediately to the left
`|`	Combine regular expressions with an OR
`\`	Use it if you want to match a literal metacharacter. For example `\.` matches an actual period, and `\?` matches an acutal question mark
`( )`	Used to group characters into a single unit to which modifiers like `*` or `+` can be applied, to delimit the extent of `|`'s (or to be used in more advanced expressions for inserting replacement groups)

We note in the above table that more explanation is needed for the concept of *character classes* defined by `[ ]`. These are similar to what you are already familiar with in terms of *globbing* file names. In short, if you put a variety of characters between square brackets, it means "match any one of these characters." For example, `/[aceACE]/` means "match any upper- or lowercase `a`, `c`, or `e`. Within those square brackets, `^` and `-` have special meaning depending on where between the square brackets they occur:

- A `-`, when *between* letters or numbers, indicates a range: `/[a-zA-Z]/` means any letter; /[0-5]/ means any numeral between 0 and 5.

- A `-` at the beginnging or end of the characters inside the square brackets just means `-`: `/[-;ab]/` means match any of the characters `-`, `;`, `a`, or `b`.
- A `^` *at the beginning of the characters inside the square brackets* negates the character class, meaning the match will be to anything *not* within the character class, i.e., `[^ABC]` means match any character that is *not* `A` nor `B` nor `C`. If the `^` is not at the beginning of the characters within the `[ ]` then it carries no special meaning: `[:;^%]` matches any of those four punctuation characters.'
- All characters, except `-` and `^` in the correct positions, and the backslash `\`, are interpreted *literally* (i.e., not as metacharacters) within the `[ ]`. So you can match any one of `?`, `*`, `(`, `)`, or `+`, for instance, with `/[?*()+]/`.

All these metacharacters might seem a little cryptic, so I provide a few examples, here, that might help to make it more clear. They are presented (i.e. "match any line") as if they are part of an `awk` test.

```
# match any line that starts with @RG
/^@RG/

# match any line that starts with @ followed by any two characters
/^@../

# match any line that starts with @ followed by any
# two uppercase letters
/^@[A-Z][A-Z]/

# the above could also be written as:
/^@[A-Z]{2}/

# This matches phone numbers formatted either like
# (###) ###-####, or ###-###-####
/\(?[0-9]{3}(\) |-)[0-9]{3}-[0-9]{4}/

# And if you can parse that out, you get an A for the day!

# match anything that starts with T456, then has any
# number (including 0) of any other characters, then
# ends with ".fq"
/T456.*\.fq/

# Note the use of backslash to escape the .

# Match either "big mess" or "huge mess"
/(big|huge) mess/
```

Now it's your turn

Here are some things to try.

```
# 1. use alternation (the |) to match lines in the SAM file data/DPCh_plate1_F12_S72.sam
# that start with either @PG or @RG


# 2. search through the fastq file data/DPCh_plate1_F12_S72.R1.fq.gz
# to find any line with exactly two consecutive occurrences of any of the following charac
# ! " # $ % &


# What are we searching for in the above?
```

4.1.4 A variety of tests

When you want to test whether a line will be processed by an action block, or not, you have many options. Among the major ones (each given with an example or two) are:

1. match a regular expression anywhere in a line:

```
awk '/^@RG/'
awk '/Fall/'
```

2. match a regular expression in a specific column/field

```
awk '$3 ~ /Sacramento/'
awk '$1 ~ /Chr[123]/'
```

3. test for equality, using ==, of a single column to a string, or a number. Note these are *not* regular expressions, but actual statements of equality. Strings in such a context are surrounded by double quotes.

```
awk '$1 == "NC_07124.1"'
awk '$5 == 100'
```

4. Use comparison operators `<`, `<=`, `>`, `>=`, and `!=` (not equals) to compare a column to a string (compared by lexicographical order) or a number (compared by numerical order)

```
awk '$2 <= "Aardvark"'
awk '$7 > 25'
```

5. test the value of a user-defined variable that may be changing as lines are getting processed.

```
awk 'n > 356'
awk 'my_word == "Loony"'
```

6. test the value of an internal variable, like `NR` or `NF`

```
awk 'NR < 25'
awk 'NF == 13'
```

As these tests are effectively things that return a value of TRUE or FALSE, they can be combined with logical AND and logical OR operators. In awk, the logical AND is `&&` and the logical OR is `||`. To make the intent of long combinations clear, and to specify specific combinations, the tests can be grouped with parentheses:

```
awk '(units = "days" && n > 356) || (units == "months" && n > 12) {print "More than a year
```

4.1.5 Code in the action blocks

Within the action blocks you write computer code to do things, and `awk` has many features you expect in a programming language.

Separate lines of code can be ended with a line return (i.e., in scripts), or they can be ended with a semicolon. Variable assignment is done with the `=` sign. Unlike the bash shell, you can have spaces around the `=`. One interesting aspect of `awk` is that the variables are *untyped*. This means that you don't have to tell `awk` ahead of time whether a variable is going to hold a number, or a string, etc. Everything is stored as a string, but when used in a numeric context, if it makes sense, the variable will be treated as a number. So, you don't have to worry too much about the type of variables.

If a value has not yet been assigned to a variable, but it is used in a numeric context, the value is assumed to be 0; if used in a string context, its value is assumed to be the empty string `""`.

You can use `for` loops within `awk`. They have the syntax of the C language:

```
for(var = initial; test; increment)
```

For example, this cycles `i` over values starting from 1, until 10, each time incrementing the value by 1:

```
for(i=1;i<=10;i++)
```

Note that the `++` means "add one to the variable to my left."

You can also increment by larger amounts. What do you think this would do?

```
echo boing | awk '{for(i=5; i<=25; i+=5) print i}'
```

If the body of the loop consists of multiple lines of code (or even just one) those lines can be *grouped* using curly braces. Here is an examle of using a `for` loop to print the columns of the Chinook sample sheet in row format:

```
awk -F"," '
  NR == 1 {for(i=1;i<=NF;i++) head[i] = $i; next}
  {for(i=1;i<=NF;i++) {
      print i ". " head[i] ": " $i;
    }
    print "----------------------"
  }
' data/wgs-chinook-samples.csv
```

Arrays are implemented as *associative arrays*. Thus, rather than being arrays with elements indexed (and accessed) by natural numbers (i.e. `array[1]`, `array[2]`, etc.), arrays are indexed by any string. This is sometimes called a hash. In python it is called a *dictionary*. So, within an awk action block you could see

```
n[$3]++
```

which means "find the element of the array `n` that is associated with the *key* $3, and then add 1 to that element". This can be quite useful for counting up the occurrences of different strings or values in a column. Of course, in order to actually see what the values are, after they have been counted up, you need

to be able to cycle over all the different *keys* of the array, and print the key and the value for each. In `awk`, you cycle over the keys of an array using

```
for (i in array)
```

where `for` and `in` are keywords, `i` is any variable name you want, and `array` is the name of an array variable. Unfortunately, you have no control over the *order* in which the different keys in the array are visited! Example:

```
# count the number of fish from different sampling collections
# in the wgs-chinook-samples.tsv file:
awk -F"\t" '
  NR > 1 {n[$6]++}
  END {for(i in n) print i ":", n[i]}
' data/wgs-chinook-samples.tsv

# gives us:
Coleman Hatchery Late Fall: 16
Feather River Hatchery Spring: 16
Feather River Hatchery Fall: 16
Salmon River Spring: 16
Trinity River Hatchery Fall: 16
Trinity River Hatchery Spring: 16
Butte Creek Spring: 16
Sacramento River Winter: 16
Salmon River Fall: 16
San Joaquin River Fall: 16
```

The above demonstrates the use of the very important `END` specifier in the test position. The `END` there means "perform the actions in the action block once all the lines of the file have been processed."

Now it's your turn!
We want to cycle through the file `DPCh_plate1_F12_S72.R1.fq.gz` and count up the number of times different base-quality sequences occur in the file. The base quality scores occur on lines 4, 8, 12, … (so, the line number divided by 4 has no remainder. Note that `x % 4` gives the remainder when `x` is divided by 4. We want the output on each line to be in the format:

```
Number_of_occurrences  Base_quality_score_sequence
```

```
# go for it!
```

After doing that, you might wish that things were sorted differently, like highest to lowest in terms of # of occurrences. Try piping the output into:

```
sort -n -b -r -k 1
```

Pipe that to less and look through it. It is actually pretty cool.

Conditional tests use `if` and `else`, with blocking by curly braces. Below we will demonstrate the use of `if` and `else` and, how they can be put together to make an `ifelse`-like construction. At the same time we demonstrate how output from `print` statements in `awk` can be redirected to files, from within the `awk` script itself. This uses much the same syntax as the shell.

Imagine that we want to put the IDs (called the `vcf_names`) of the fish in `data/wgs-chinook-samples.csv` into separate files, one for each of the `run_types` of "Fall", "Winter", and "Spring", and another file for anything else. We could do that like this:

```
awk -F"," '
  NR > 1 {
    if($NF == "Fall") {
      print $1, $NF > "fall.txt"
    } else if($NF == "Winter") {
      print $1, $NF > "winter.txt"
    } else if($NF == "Spring") {
      print $1, $NF > "spring.txt"
    } else {
      print $1, $NF > "other.txt"
    }
  }
' data/wgs-chinook-samples.csv
```

Now look at the four files produced.

NB: Parsing CSV files with `awk` is not always so straightforward as doing `-F","` because the CSV specification allows for commas that do not separate fields to be hidden within quotation marks. Don't expect to parse complex CSV files made by Excel, for example, to be parsed this easily with `awk`. It is better to save them as TAB separate files, typically.

Mathematical operations: `awk` has got them all: `+`, `-`, `*`, `/`, `%`, as well as "operate and reassign" versions: `+=`, `-=`, `*=`, `/=`, as well as: `exp`, `log`, `sqrt`, `sin`, `cos`, and `atan2`

Built in functions: `awk` has a limited set of built-in functions. See `man awk` for a full listing. The ones I find I use all the time are:

- `length(n)` : return the number of characters of the variable `n`
- `substr(s, m, n)` : return the portion of string `s` that begins at position `m` (counted starting from 1), and is `n` characters long

- `sub(r, t, s)` : substitute string `t` for the first occurrence of the regular expression `r` in the string `s`. If `s` is not given, `$0` is used.
- `gsub` : just like `sub` but replaces *all* (rather than just the first) occurrences of the regular expression `r` in string `s`.
- `split(s, a, fs)` : split the string `s` into an array variable named `a` at occurrences of the regular expression `fs`. The function returns the number of pieces the string was split into. Afterward each part of the string can be accessed like `a[1]`, `a[2]`, and so forth. (Good example = splitting out fields from a column in a VCF file.)

Take control of output formatting with the `printf()` function:
I still need to write this.

4.1.6 Using `awk` to assign to shell variables

We leave our discussion of `awk` by noting that its terse syntax makes it a perfect tool for creating shell variables that we can do things (like cycling over them) with.

For example, imagine we wanted to do something to all the FASTQ files associated with the fish in `data/wgs-chinook-samples.csv` that are of `run_type` = "Winter". We can get all their IDs using command substitution and then cycle over them in the shell like this:

```
WINTERS=$(awk -F"," '$NF == "Winter" {print $1}' data/wgs-chinook-samples.csv)

for i in $WINTERS; do
  echo "FASTQS are: $i.R1.fq.gz     $i.R2.fq.gz"
done
```

4.1.7 Passing Variables into `awk` with `-v`

Gotta let people know about this!

4.1.8 Writing awk scripts in files

Need to do this.

4.2 sed

`sed` is a "stream editor". Though it is capable of all sorts of things, to be honest, I use it almost exclusively to do simple find-and-replace operations. The syntax for that is:

```
sed 's/regex/replacement/g;' file
```

where `regex` is a regular expression and `replacement` is the string you wish to replace any segments of `file` that match `regex`. The `s` means "substitute" and is like a command to `sed`, and the `g` means "globally", without which only the first match of `regex` on each line would be replaced.

Multiple instances of the "s" command can be given, like:

```
sed '
  s/regex1/replacement1/g;
  s/regex2/replacement2/g;
  s/regex3/replacement3/g;
' file
```

The separate commands are done in order, line by line.

Now it's your turn!
Take the output of:

```
WINTERS=$(awk -F"," '$NF == "Winter" {print $1}' data/wgs-chinook-samples.csv)

for i in $WINTERS; do
  echo "FASTQS are: $i.R1.fq.gz     $i.R2.fq.gz"
done
```

and pipe it into `sed` to changes the `R1` and `R2` into `r1` and `r2`, and to remove the `.gz` from the end of each file name. Remember to backslash-escape that period!

5

Working on remote servers

5.1 Accessing remote computers

The primary protocol for accessing remote computers in this day and age is `ssh` which stands for "Secure Shell." In the protocol, your computer and the remote computer talk to one another and then choose to have a "shared secret" which they can use as a key to encrypt data traffic from one to the other. The amazing thing is that the two computers can actually tell each other what that shared secret is by having a conversation "in the open" with one another. That is a topic for another day, but if you are interested, you could read about it here[1].

At any rate, the SSH protocal allows for secure access to a remote server. It involves using a username and a password, and, in many cases today, some form of two-factor authentication (i.e., you need to have your phone involved, too!). Different remote servers have different routines for logging in to them, and they are also all configured a little differently. The main servers we are concerned about in these teaching materials are:

1. The Hummingbird cluster at UCSC, which is accessible by anyone with a UCSC blue username/password.
2. The Summit Supercomputer at CU Boulder which is accessible by all graduate students and faculty at CSU.
3. The Sedna cluster housed at the National Marine Fisheries Service, Northwest Fisheries Science Center. This is accessible only by those Federal NMFS employees whom have been granted access.

Happily, all of these systems use SLURM for job scheduling (much more about that in the next chapter); however are a few vagaries to each of these systems that we will cover below.

[1] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

5.1.1 Windows

If you are on a Windows machine, you can use the `ssh` utility from your Git Bash shell, but that is a bit of a hassle from RStudio. And a better terminal emulator is available if you are going to be accessing remote computers. It is recommended that you install and use the program PuTTy[2]. The steps are pretty self-explanatory and well documented. Instead of using `ssh` on a command line you put a host name into a dialog box, etc.

5.1.2 Hummingbird

Directions for UCSC students and staff to login to Hummingbird are available at https://www.hb.ucsc.edu/getting-started/. If you are not on the UCSC campus network, you need to use the UCSC VPN to connect.

By default, this cluster uses `tcsh` for a shell rather than `bash`. To keep things consistent with what you have learned about `bash`, you will want to automatically switch to `bash` upon login. You can do this by adding a file `~/.tcshrc` whose contents are:

```
setenv SHELL /usr/bin/bash
exec /usr/bin/bash --login
```

Then, configure your `bash` environment with your `~/.bashrc` and `~/.bash_profile` as described in Chapter 2.5.

The `tmux` settings (See section 5.3) in hummingbird are a little messed up as well, making it hard to set window names that don't get changed the moment you make another command. Therefore, you must make a file called `~/.tmux.conf` and put this line in it:

```
set-option -g allow-rename off
```

5.1.3 Summit

To get an account on Summit, see https://www.acns.colostate.edu/hpc/summit-get-started/. Account creation is automatic for graduate students and faculty. This setup requires that you get an app called Duo on your phone for doing two-factor authentication.

Instructions for logging into Summit are at https://www.acns.colostate.edu/hpc/#remote-login.

[2] https://www.ssh.com/ssh/putty

On your local machine (i.e., laptop), you might consider adding an alias to your `.bashrc` that will let you type `summit` to issue the login command. For example:

```
alias summit='ssh csu_eid@colostate.edu@login.rc.colorado.edu'
```

where you replace `csu_eid` with your actual CSU eID.

5.1.4 Sedna

To connect to this cluster you must be on the NMFS network, or connected to it via the VPN, then `ssh` with, for example:

`ssh eanderson@sedna.nwfsc2.noaa.gov`

but using your own username.

5.2 Transferring files to remote computers

5.2.1 `sftp` (via `lftp`)

Most Unix systems have a command called `scp`, which works like `cp`, but which is designed for copying files to and from remote servers using the SSH protocol for security. This works really well if you have set up a public/private key pair to allow SSH access to your server without constantly having to type in your password. Use of public-private keypairs is unfortunately, not an option (as far as I can tell) on new NSF-funded clusters that use 2-factor authentication (like SUMMIT at CU Boulder). Trying to use `scp` in such a context becomes an endless cycle of entering your password and checking your phone for a DUO push. Fortunately, there are alternatives.

The administrators of the SUMMIT supercomputer at CU Boulder recommend the `sftp` utility for transferring files from your laptop to the server. This works reasonably well. The syntax for a CSU student or affiliate connecting to the server is

```
sftp csu_userEID@colostate.edu@login.rc.colorado.edu

# for example, here is mine:
sftp eriq@colostate.edu@login.rc.colorado.edu
```

After doing this you have to give your eID password followed by `,push`, and then approve the DUO push request on your phone. Once that is done, you have a "line open" to the server and can use the commands of `sftp` to transfer files around. However, the vanilla version of `sftp` (at least on a Mac) is unbelievably limited, because there is simply no good support for TAB completion within the utility for navigating directories on the server or upon your laptop. It must have been developed by troglodytes…consequently, I won't describe vanilla `sftp` further.

If you are on Windows, it looks like the makers of PuTTY also bring you PSFTP[3] which might be useful for you for file transfer. Go for it!

If you are on a Mac, you can install `lftp` (`brew install lftp`: note that I need to write a section about installing command line utilities via homebrew somewhere in this handbook). `lftp` provides the sort of TAB completion of paths that you, by now, will have come to know and love and expect.

Before you connect to your server with `lftp` there are a few customizations that you will want to do in order to get nicely colored output, and to avoid having to login repeatedly during your `lftp` session. You must make a file on your laptop called `~/.lftprc` and put the following lines in it:

```
set color:dir-colors "rs=0:di=01;36:fi=01;32:ln=01;31:*.txt=01;35:*.html=00;35:"
set color:use-color true

set net:idle 5h
set net:timeout 5h
```

Now, to connect to SUMMIT with `lftp`, you use this syntax (shown for my username):

```
lftp sftp://eriq@colostate.edu@login.rc.colorado.edu
```

That can be a lot to type, so I would recommend putting something this in your `.bashrc`:

```
alias summit_ftp='lftp sftp://eriq@colostate.edu@login.rc.colorado.edu'
```

so you can just type `summit_ftp` (which will TAB complete…) to launch that command.

After you issue that command, you put in your password (on SUMMIT, followed by `,push`). `lftp` then caches your password, and will re-issue it, if necessary, to execute commands. It doesn't actually send your password until

[3] https://www.ssh.com/ssh/putty/putty-manuals/0.68/Chapter6.html#psftp

you try a command like `cls`. On the SUMMIT system, with the default `lftp` settings, after 3 minutes of idle time, when you issue an `sftp` command on the server, you will have to approve access with the DUO app on your phone again. However, the line last two lines in the `~/.lftprc` file listed above ensure that your connection to SUMMIT will stay active even through 5 hours of idle time, so you don't have to keep clicking DUO pushes on your phone. After 5 hours, if you try issuing a command to the server in `lftp`, it will use your cached password to reconnect to the server. On SUMMIT, this means that you only need to deal with approving a DUO push again—not re-entering your password. If you are working on SUMMIT daily, it makes sense to just keep one Terminal window open, running `lftp`, all the time.

Once you have started your `lftp/sftp` session this way, there are some important things to keep in mind. The most important of which is that the `lftp` session you are in maintains a *current working directory* on both the server and on your laptop. We will call these the *server working directory* and the *laptop working directory*, respectively, (Technically, we ought to call the laptop working directory the *client working directory* but I find that is confusing for people, we we will stick with *laptop*.) There are two different commands to see what each current working directory is:

- `pwd` : print the *server working directory*
- `lpwd` : print *laptop working directory* (the preceding `l` stands for *local*).

If you want to change either the server or the laptop current working directory you use:

- `cd` *path* : change the server working directory to *path*
- `lcd` *path* : change the laptop working directory to *path*.

Following `lcd`, TAB-completion is done for paths *on the laptop*, while following `cd`, TAB-completion is done for paths *on the server*.

If you want to list the contents of the different directories *on the servers* you use:

- `cls` : list things in the server working directory, or
- `cls` *path* : list things in *path* on the server.

Note that `cls` is a little different than the `ls` command that comes with `sftp`. The latter command always prints in long format and does not play nicely with colorized output. By contrast, `cls` is part of `lftp` and it behaves mostly like your typical Unix `ls` command, taking options like `-a`, `-l` and `-d`, and it will even do `cls -lrt`. Type `help cls` at the `lftp` prompt for more information.

If you want to list the contents of the different directories on your laptop, you use `ls` *but you preface it with a* `!`, which means "execute the following on my laptop, not the server." So, we have:

- `!ls` : list the contents of the laptop working directory.

- `!ls` *path* : list the contents of the laptop path *path*.

When you use the `!` at the beginning of the line, then all the TAB completion occurs in the context of the laptop current working directory. Note that with the `!` you can do all sorts of typical shell commands on your laptop from within the `lftp` session. For example `!mkdir this_on_my_laptop` or `!cat that_file`, etc.

If you wish to make a directory on the *server*, just use `mkdir`. If you wish to remove a file from the server, just use `rm`. The latter works much like it does in bash, but does not seem to support globbing (use `mrm` for that!) In fact, you can do a lot of things (like `cat` and `less`) on the server *as if you had a bash shell running on it* through an SSH connection. Just type those commands at the `lftp` prompt.

5.2.1.1 Transferring files using `lftp`

To this point, we haven't even talked about our original goal with `lftp`, which was to *transfer files from our laptop to the server* or from *the server to our laptop*. The main `lftp` commands for those tasks are: `get`, `put`, `mget`, `mput`, and `mirror`—it is not too much to have to remember.

As the name suggests, `put` is for *putting* files from your laptop onto the server. By default it puts files into the server working directory. Here is an example:

```
put laptopFile_1 laptopFile_2
```

If you want to put the file into a different directory on the server (that must already exist) you can use the `-O` option:

```
put -O server_dest_dir laptopFile_1 laptopFile_2
```

The command `get` works in much the same way, but in reverse: you are *getting* things *from the server to your laptop*. For example:

```
# copy to laptop working directory
get serverFile_1 serverFile1_2

# copy to existing directory laptop_dest_dir
get -O laptop_dest_dir serverFile_1 serverFile1_2
```

Neither of the commands `get` or `put` do any of the pathname expansion (or "globbing" as it we have called it) that you will be familiar with from the `bash` shell. To effect that sort of functionality you must use `mput` and `mget`, which, as the `m` prefix in the command names suggests, are the "multi-file" versions

of `put` and `get`. Both of these commands also take the -O option, if desired, so that the above commands could be rewritten like this:

```
mput -O server_dest_dir laptopFile_[12]
# and
mget -O laptop_dest_dir serverFile_[12]
```

Finally, there is not a *recursive* option, like there is with `cp`, to any of `get`, `put`, `mget`, or `mput`. Thus, you cannot use any of those four to put/get entire directories on/from the server. For that purpose, `lftp` has reserved the `mirror` command. It does what it sounds like: it mirrors a directory from the server to the laptop. The `mirror` command can actually be used in a lot of different configurations (between two remote servers, for example) and with different settings (for example to change only pre-existing files older than a certain date). However, here, we will demonstrate only its common use case of copying directories between a server and laptop here.

To copy a directory `dir`, and its contents, from your server to your laptop current directory you use:

```
mirror dir
```

To copy a directory `ldir` from your laptop to your server current directory you use `-R` which transmits the directory in the reverse direction:

```
mirror -R ldir
```

Learning to use `lftp` will require a little bit more of your time, but it is worth it, allowing you to keep a dedicated terminal window open for file transfers with sensible TAB-completion capability.

5.2.2 git

Most remote servers you work on will have `git` by default. If you are doing all your work on a project within a single repository, you can use `git` to keep scripts and other files version-controlled on the server. You can also push and pull files (not big data or output files!) to GitHub, thus keeping things backed up and version controlled, and providing a useful way to synchronize scripts and other files in your project between the server and your laptop.

Example:

1. write and test scripts on your laptop in a repo called `my-project`

2. commit scripts on your laptop and push them to GitHub in a repo also called `my-project`
3. pull `my-project` from GitHub to the server.
4. Try running your scripts in `my-project` on your server. In the process, you may discover that you need to change/fix some things so they will run correctly on the server. Fix them!
5. Once things are fixed and successfully running on the server, commit those changes and push them to GitHub.
6. Update the files on your laptop so that they reflect the changes you had to make on the server, by pulling `my-project` from GitHub to your laptop.

5.2.2.1 Configuring git on the remote server

In order to make this sort of worklow successful, you first need to ensure that you have set up git on your remote server. Doing so involves:

1. establishing your name and email that will be used with your git commits made from the server.
2. Ensuring that git password caching is set up so you don't always have to type your GitHub password when you push and pull.
3. configuring your git text editor to be something that you know how to use.

It can be useful give yourself a git name on the server that reflects the fact that the changes you are committing were made on the server.

For example, for my own setup on the Summit cluster at Boulder, I might do my git configurations by issuing these commands on the command line on the server:

```
git config --global user.name "Eric C. Anderson (From Summit)"
git config --global user.email eriq@rams.colostate.edu
git config --global credential.helper cache
git config --global core.editor nano
```

In all actuality, I would set my editor to be `vim` or `emacs`, because those are more powerful editors and I am familiar with then; however, if you are new to Unix, then `nano` is an easy-to-use editor.

You should set configurations on your server appropriate to yourself (i.e., with your name and email and preferred text editor). Once these configurations are set, you are ready to start cloning repositories from GitHub and then pushing and pulling them, as well.

To this point, we have always done those actions from within RStudio. On a remote server, however, you will have to do all these actions from the command line. That is OK, it just requires learning a few new things.

The first, and most important, issue to understand is that if you want to push new changes back to a repository that is on your GitHub account, GitHub needs to know that you have privileges to do so. One way to do this is by making sure, when you initially clone the repository, that you do so whilst letting GitHub know that you are the user that "owns" your GitHub account. This is done by inserting your GitHub username into your `git clone` request with the following syntax:

```
git clone https://username@github.com/username/repository.git
```

For example, if I want to `clone` a repository whose URL is:

```
https://github.com/eriqande/alignment-play
```

I would issue this command:

```
git clone https://eriqande@github.com/eriqande/alignment-play
```

This clones the repository to my server, and makes a note of the fact that you did so as the GitHub user `eriqande`, so that if I try to push something back to that GitHub repository, it will ask me for me proper `eriqande` GitHub password.

5.2.2.2 Using git on the remote server

When on the server, you don't have the convenient RStudio interface to git, so you have to use git commands on the command line. Fortunately these provide straightforward, command-line analogies to the RStudio GUI git interface you have become familiar with.

Intead of having an RStudio Git panel that shows you files that are new or have been modified, etc., you use `git status` in your repo to give a text report of the same.

For example, imagine that Figure 5.1 shows an RStudio project Git window describing the status of files in the repository.

That view is merely showing you a graphical view of the output of the `git status` command run at the top level of the repository which looks like this:

```
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

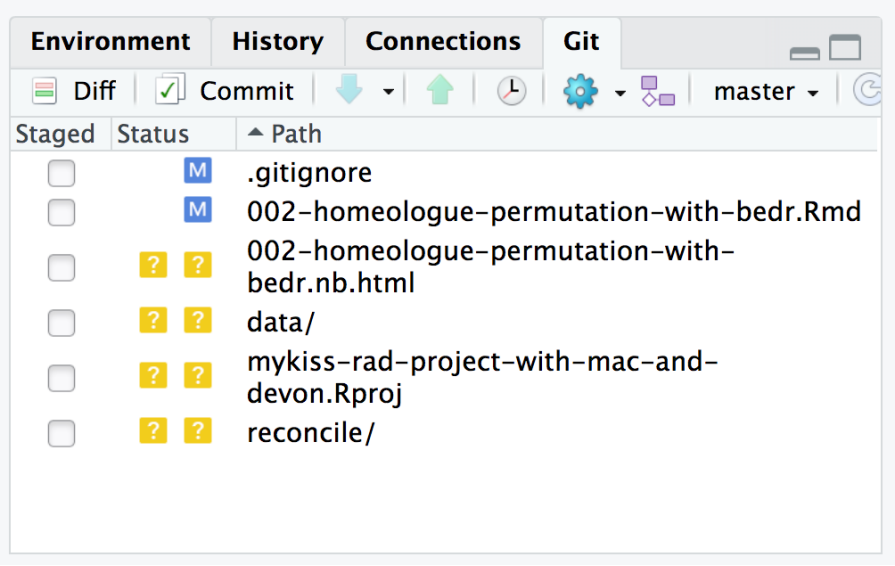


FIGURE 5.1: An example of what an RStudio git window might look like.

```
    modified:   .gitignore
    modified:   002-homeologue-permutation-with-bedr.Rmd

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    002-homeologue-permutation-with-bedr.nb.html
    data/
    mykiss-rad-project-with-mac-and-devon.Rproj
    reconcile/

no changes added to commit (use "git add" and/or "git commit -a")
```

Aha! Be sure to read that and understand that the output tells you which files are tracked by git and Modified (blue M in RStudio) and which are untracked (Yellow ? in RStudio).

Of you wanted to see a report of the changes in the files relative to the currently committed version, you could use `git diff`, passing it the file name as an argument. We will see an example of that below…

Now, recall, that in order to commit files to `git` you first must *stage* them. In RStudio you do that by clicking the little button to the left

of the file or directory in the Git window. For example, if we clicked the buttons for the `data/` directory, as well as for `.gitignore` and `002-homeologue-permutation-with-bedr.Rmd`, we would have staged them and it would look like Figure 5.2.

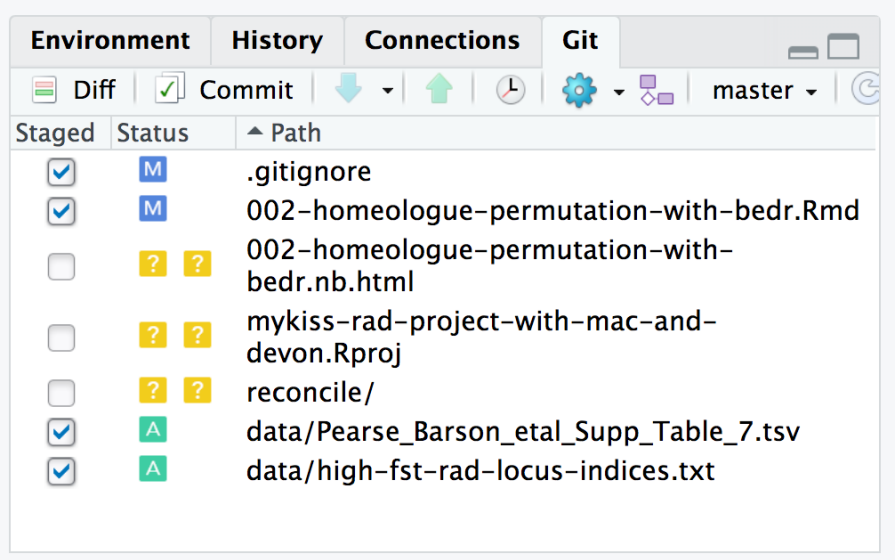


FIGURE 5.2: An example of what an RStudio git window might look like.

In order to do the equivalent operations with `git` on the command line you would use the `git add` command, explicitly naming the files you wish to *stage* for committing:

```
git add .gitignore 002-homeologue-permutation-with-bedr.Rmd data
```

Now, if you check `git status` you will see:

```
% git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   .gitignore
    modified:   002-homeologue-permutation-with-bedr.Rmd
    new file:   data/Pearse_Barson_etal_Supp_Table_7.tsv
    new file:   data/high-fst-rad-locus-indices.txt
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    002-homeologue-permutation-with-bedr.nb.html
    mykiss-rad-project-with-mac-and-devon.Rproj
    reconcile/
```

It tells you which files are ready to be committed!

In order to commit the files to git you do:

```
git commit
```

And then, to push them back to GitHub (if you cloned this repository from GitHub), you can simply do:

```
git push origin master
```

That syntax is telling git to push the `master` branch (which is the default branch in a git repository), to the repository labeled as `origin`, which will be the GitHub repository if you cloned the repository from GitHub. (If you are working with a different git branch than master, you would need to specify its name here. That is not difficult, but is beyond the scope of this chapter.)

Now, assuming that we cloned the `alignment-play` repository to our server, here are the steps involved in editing a file, committing the changes, and then pushing them back to GitHub. The command in the following is written as `[alignment-play]--%` which is telling us that we are in the `alignment-play` repository.

```
# check git status
[alignment-play]--% git status

# On branch master
nothing to commit, working directory clean

# Aha! That says nothing has been modified.
# But, now we edit the file alignment-play.Rmd
[alignment-play]--% nano alignment-play.Rmd

# In this case I merely added a line to the YAML header.

# Now, check status of the files:
```

```
[alignment-play]--% git status

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   alignment-play.Rmd
#
no changes added to commit (use "git add" and/or "git commit -a")

# We see that the file has been modified.

# Now we can use git diff to see what the changes were
[alignment-play]--% git diff alignment-play.Rmd

diff --git a/alignment-play.Rmd b/alignment-play.Rmd
index 9f75ebb..b389fae 100644
--- a/alignment-play.Rmd
+++ b/alignment-play.Rmd
@@ -3,6 +3,7 @@ title: "Alignment Play!"
 output:
   html_notebook:
     toc: true
+    toc_float: true
 ---

# The output above is a little hard to parse, but it shows
# the line that has been added: "  toc_float: true" with a
# "+" sign.

# In order to commit the changes, we do:
[alignment-play]--% git add alignment-play.Rmd
[alignment-play]--% git commit

# after that, we are bumped into the nano text editor
# to write a short message about the commit. After exiting
# from the editor, it tells us:
[master 001e650] yaml change
 1 file changed, 1 insertion(+)

# Now, to send that new commit to GitHub, we use git push origin master
[alignment-play]--% git push origin master
Password for 'https://eriqande@github.com':
```

```
Counting objects: 5, done.
Delta compression using up to 24 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 325 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://eriqande@github.com/eriqande/alignment-play
   0c1707f..001e650  master -> master
```

When pushing to a GitHub repository for the first time from your remote server you will be asked for your GitHub password. Once you have successfully provided your password, if you set password caching up correctly, then you should not have to provide your password for subsequent pushes.

Finally, if after pushing those changes to GitHub, we then pull them down to our laptop, and make more changes on top of them and push those back to GitHub, we can retrieve from GitHub to the server those changes we made on our laptop with `git pull origin master`. In other words, from the server we simply issue the command:

```
[alignment-play]--% git pull origin master
```

5.2.3 Globus

5.2.4 Interfacing with "The Cloud"

Increasingly, data scientists and tech companies alike are keeping their data "in the cloud." This means that they pay a large tech firm like Amazon, Dropbox, or Google to store their data for them in a place that can be accessed via the internet. There are many advantages to this model. For one thing, the company that serves the data often will create multiple copies of the data for backup and redundancy: a fire in a single data center is not a calamity because the data are also stored elsewhere, and can often be accessed seamlessly from those other locations with no apparent disruption of service. For another, companies that are in the business of storing and serving data to multiple clients have data centers that are well-networked, so that getting data onto and off of their storage systems can be done very quickly over the internet by an end-user with a good internet connection.

Five years ago, the idea of storing next generation sequencing data might have sounded a little crazy—it always seemed a laborious task getting the data off of the remote server at the sequencing center, so why not just keep the data in-

house once you have it? To be sure, keeping a copy of your data in-house still can make sense for long-term data archiving needs, but, today, cloud storage for your sequencing data can make a lot of sense. A few reasons are:

1. Transferring your data from the cloud to the remote HPC system that you use to process the data can be very fast.
2. As above, your data can be redundantly backed up.
3. If your institution (university, agency, etc.) has an agreement with a cloud storage service that provides you with unlimited storage and free network access, then storing your sequencing data in the cloud will cost considerably less than buying a dedicated large system of hard drives for data backup. (One must wonder if service agreements might not be at risk of renegotiation if many researchers start using their unlimited institutional cloud storage space to store and/or archive their next generation sequencing data sets. My own agency's contract with Google runs through 2021…but I have to think that these services are making plenty of money, even if a handful of researchers store big sequence data in the cloud. Nonetheless, you should be careful not to put multiple copies of data sets, or intermediate files that are easily regenerated, up in the cloud.)
4. If you are a PI with many lab members wishing to access the same data set, or even if you are just a regular Joe/Joanna researcher but you wish to share your data, it is possible to effect that using your cloud service's sharing settings. We will discuss how to do this with Google Drive.

There are clearly advantages to using the cloud, but one small hurdle remains. Most of the time, working in an HPC environment, we are using Unix, which provides a consistent set of tools for interfacing with other computers using SSH-based protocols (like `scp` for copying files from one remote computer to another). Unfortunately, many common cloud storage services do not offer an SSH based interface. Rather, they typically process requests from clients using an HTTPS protocol. This protocol, which effectively runs the world-wide web, is a natural choice for cloud services that most people will access using a web browser; however, Unix does not traditionally come with a utility or command to easily process the types of HTTPS transactions needed to network with cloud storage. Furthermore, there must be some security when it comes to accessing your cloud-based storage—you don't want everyone to be able to access your files, so your cloud service needs to have some way of authenticating people (you and your labmates for example) that are authorized to access your data.

These problems have been overcome by a utility called `rclone`, the product of a comprehensive open-source software project that brings the functionality of the `rsync` utility (a common Unix tool used to synchronize and mirror

file systems) to cloud-based storage. (Note: `rclone` has nothing to do with the R programming language, despite its name that looks like an R package.) Currently `rclone` provides a consistent interface for accessing files from over 35 different cloud storage providers, including Box, Dropbox, Google Drive, and Microsoft OneDrive. Binaries for `rclone` can be downloaded for your desktop machine from https://rclone.org/downloads/. We will talk about how to install it on your HPC system later.

Once `rclone` is installed and in your `PATH`, you invoke it in your terminal with the command `rclone`. Before we get into the details of the various `rclone` subcommands, it will be helpful to take a glance at the information `rclone` records when it configures itself to talk to your cloud service. To do so, it creates a file called `~/.config/rclone/rclone.conf`, where it stores information about all the different connections to cloud services you have set up. For example, that file on my system looks like this:

```
[gdrive-rclone]
type = drive
scope = drive
root_folder_id = 1I2EDV465N5732Tx1FFAiLWOqZRJcAzUd
token = {"access_token":"bs43.94cUFOe6SjjkofZ","token_type":"Bearer","refresh_token":"1/Mr
client_id = 2934793-oldk97lhld88dlkh301hd.apps.googleusercontent.com
client_secret = MMq3jdsjdjgKTGH4rNV_y-NbbG
```

In this configuration:

- `gdrive-rclone` is the name by which rclone refers to this cloud storage location
- `root_folder_id` is the ID of the Google Drive folder that can be thought of as the root directory of `gdrive-rclone`. This ID is not the simple name of that directory on your Google Drive, rather it is the unique name given by Google Drive to that directory. You can see it by navigating in your browser to the directory you want and finding it after the last slash in the URL. For example, in the above case, the URL is: `https://drive.google.com/drive/u/1/folders/1I2EDV465N5732Tx1FFAiLWOqZRJcAzUd`
- `client_id` and `client_secret` are like a username and a shared secret that `rclone` uses to authenticate the user to Google Drive as who they say they are.
- `token` are the credentials used by `rclone` to make requests of Google Drive on the basis of the user.

Note: the above does not include my real credentials, as then anyone could use them to access my Google Drive!

To set up your own configuration file to use Google Drive, you will use the `rclone config` command, but before you do that, you will want to wrangle a client_id from Google. Follow the directions at https://rclone.org/drive/#making-your-own-client-id. Things are a little different from in their step

by step, but you can muddle through to get to a screen with a client_ID and a client secret that you can copy onto your clipboard.

Once you have done that, then run `rclone config` and follow the prompts. A typical session of `rclone config` for Google Drive access is given here[4]. Don't choose to do the advanced setup; however do use "auto config," which will bounce up a web page and let you authenticate rclone to your Google account.

It is worthwhile first setting up a config file on your laptop, and making sure that it is working. After that, you can copy that config file to other remote servers you work on and immediately have the same functionality.

5.2.4.1 Encrypting your config file

While it is a powerful thing to be able to copy a config file from one computer to the next and immediately be able to access your Google Drive account. That might (and should) also make you a little bit uneasy. It means that if the config file falls into the wrong hands, whoever has it can gain access to everything on your Google Drive. Clearly this is not good. Consequently, once you have created your rclone config file, and well before you transfer it to another computer, you must encrypt it. This makes sense, and fortunately it is fairly easy: you can use `rclone config` and see that encryption is one of the options. When it is encrypted, use `rclone config show` to see what it looks like in clear text.

The downside of using encryption is that you have to enter your password every time you make an rclone command, but it is worth it to have the security.

Here is what it looks like when choosing to encrypt one's config file:

```
% rclone config
Current remotes:

Name                 Type
====                 ====
gdrive-rclone        drive

e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
```

[4] https://rclone.org/drive/

```
q) Quit config
e/n/d/r/c/s/q> s
Your configuration is not encrypted.
If you add a password, you will protect your login information to cloud services.
a) Add Password
q) Quit to main menu
a/q> a
Enter NEW configuration password:
password:
Confirm NEW configuration password:
password:
Password set
Your configuration is encrypted.
c) Change Password
u) Unencrypt configuration
q) Quit to main menu
c/u/q> q
Current remotes:

Name                 Type
====                 ====
gdrive-rclone        drive

e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> q
```

Once that file is encrypted, you can copy it to other machines for use.

5.2.4.2 Basic Maneuvers

The syntax for use is:

```
rclone [options] subcommand  parameter1 [parameter 2...]
```

The "subcommand" part tells `rclone` what you want to do, like `copy` or `sync`, and the "parameter" part of the above syntax is typically a path specification to a directory or a file. In using rclone to access the cloud there is not a root di-

rectory, like `/` in Unix. Instead, each remote cloud access point is treated as the root directory, and you refer to it by the name of the configuration followed by a colon. In our example, `gdrive-rclone:` is the root, and we don't need to add a `/` after it to start a path with it. Thus `gdrive-rclone:this_dir/that_dir` is a valid path for `rclone` to a location on my Google Drive.

Very often when moving, copying, or syncing files, the parameters consist of:

```
source-directory  destination-directory
```

One very important point is that, unlike the Unix commands `cp` and `mv`, rclone likes to operate on directories, not on multiple named files.

A few key subcommands:

- `ls`, `lsd`, and `lsl` are like `ls`, `ls -d` and `ls -l`

```
rclone  lsd gdrive-rclone:
rclone  lsd gdrive-rclone:NOFU
```

- `copy`: copy the *contents* of a source *directory* to a destination *directory*. One super cool thing about this is that `rclone` won't re-copy files that are already on the destination and which are identical to those in the source directory.

```
rclone copy bams gdrive-rclone:NOFU/bams
```

Note that the destination directory will be created if it does not already exist.
- `sync`: make the contents of the destination directory look just like the contents of the source directory. *WARNING* This will delete files in the destination directory that do not appear in the source directory.

A few key options:

- `--dry-run`: don't actually copy, sync, or move anything. Just tell me what you would have done.
- `--progress`: give me progress information when files are being copied. This will tell you which file is being transferred, the rate at which files are being transferred, and and estimated amount of time for all the files to be transferred.
- `--tpslimit 10`: don't make any more than 10 transactions a second with Google Drive (should always be used when transferring files)
- `---fast-list`: combine multiple transactions together. Should always be used with Google Drive, especially when handling lots of files.
- `--drive-shared-with-me`: make the "root" directory a directory that shows all of the Google Drive folders that people have shared with you. This is key for accessing folders that have been shared with you.

For example, try something like:

```
rclone --drive-shared-with-me lsd gdrive-rclone:
```

Important Configuration Notes!! Rather than always giving the `--progress` option on the command line, or always having to remember to use `--fast-list` and `--tps-limit 10` (and remember what they should be…), you can set those options to be invoked "by default" whenever you use rclone. The developers of `rclone` have made this possible by setting *environment variables* in your `~/.bashrc`.

If you have an rclone option called `--fast-limit`, then the corresponding environment variable is named `RCLONE_FAST_LIMIT`—basically, you start with `RCLONE_` then you just drop the first two dashes of the option name, replace the remaining dashes with underscores, and turn it all into uppercase to make the environment variable. So, you should, at a minimum add these lines to your `~/.bashrc`:

```
# Environment variables to use with rclone/google drive always
export RCLONE_TPS_LIMIT=10
export RCLONE_FAST_LIST=true
export RCLONE_PROGRESS=true
```

5.2.4.3 filtering: Be particular about the files you transfer

`rclone` works a little differently than the Unix utility `cp`. In particular, `rclone` is not set up very well to copy individual files. While there is a an `rclone` command known as `copyto` that will allow you copy a single file, you cannot (apparently) specify multiple, individual files that you wish to copy.

In other words, you can't do:

```
rclone copyto this_file.txt that_file.txt another_file.bam gdrive-rclone:dest_dir
```

In general, you will be better off using `rclone` to copy the *contents* of a directory to the inside of the destination directory. However, there are options in `rclone` that can keep you from being totally indiscriminate about the files you transfer. In other words, you can *filter* the files that get transferred. You can read about that at https://rclone.org/filtering/.

For a quick example, imagine that you have a directory called `Data` on you Google Drive that contains both VCF and BAM files. You want to get only the VCF files (ending with `.vcf.gz`, say) onto the current working directory on your cluster. Then something like this works:

```
rclone copy --include *.vcf.gz gdrive-rclone:Data ./
```

Note that, if you are issuing this command on a Unix system in a directory where the pattern `*.vcf.gz` will expand (by globbing) to multiple files, you will get an error. In that case, wrap the pattern in a pair of single quotes to keep the shell from expanding it, like this:

```
rclone copy --include '*.vcf.gz' gdrive-rclone:Data ./
```

5.2.4.4 Feel free to make lots of configurations

You might want to configure a remote for each directory-specific project. You can do that by just editing the configuration file. For example, if I had a directory deep within my Google Drive, inside a chain of folders that looked like, say, `Projects/NGS/Species/Salmon/Chinook/CentralValley/WinterRun` where I was keeping all my data on a project concerning winter-run Chinook salmon, then it would be quite inconvenient to type `Projects/NGS/Species/Salmon/Chinook/CentralValley/WinterRun` every time I wanted to copy or sync something within that directory. Instead, I could add the following lines to my configuration file, essentially copying the existing configuration and then modifying the configuration name and the `root_folder_id` to be the Google Drive identifier for the folder `Projects/NGS/Species/Salmon/Chinook/CentralValley/WinterRun` (which one can find by navigating to that folder in a web browser and pulling the ID from the end of the URL.) The updated configuration could look like:

```
[gdrive-winter-run]
type = drive
scope = drive
root_folder_id = 1MjOrclmP1udhxOTvLWDHFBVET1dF6CIn
token = {"access_token":"bs43.94cUFOe6SjjkofZ","token_type":"Bearer","refresh_token":"1/Mr
client_id = 2934793-oldk97lhld88dlkh301hd.apps.googleusercontent.com
client_secret = MMq3jdsjdjgKTGH4rNV_y-NbbG
```

As long as the directory is still within the same Google Drive account, you can re-use all the authorization information, and just change the `[name]` part and the `root_folder_id`. Now this:

```
rclone copy src_dir gdrive-winter-run:
```

puts items into `Projects/NGS/Species/Salmon/Chinook/CentralValley/WinterRun` on the Google Drive without having to type that God-awful long path name.

5.2.4.5 Installing rclone on a remote machine without sudo access

The instructions on the website require root access. You don't have to have root access to install rclone locally in your home directory somewhere. Copy the download link from https://rclone.org/downloads/ for the type of operating system your remote machine uses (most likely Linux if it is a cluster). Then transfer that with `wget`, unzip it and put the binary in your PATH. It will look something like this:

```
wget https://downloads.rclone.org/rclone-current-linux-amd64.zip
unzip rclone-current-linux-amd64.zip
cp rclone-current-linux-amd64/rclone ~/bin
```

You won't get manual pages on your system, but you can always find the docs on the web.

5.2.4.6 Setting up configurations on the remote machine...

Is as easy as copying your config file to where it should go, which is easy to find using the command:

```
rclone config file
```

5.2.4.7 Some other usage tips

Following an email exchange with Ren, I should mention how to do an md5 checksum on the remote server to make sure that everything is correctly there.

5.2.5 Getting files from a sequencing center

Very often sequencing centers will post all the data from a single run of a machine at a secured (or unsecured) http address. You will need to download those files to operate on them on your cluster or local machine. However some of the files available on the server will likely belong to other researchers and you don't want to waste time downloading them.

Let's take an example. Suppose you are sent an email from the sequencing center that says something like:

> Your samples are AW_F1 (female) and AW_M1 (male). You should be able to access the data from this

link provided by YCGA: http://sysg1.cs.yale.edu:3010/5lnO9bs3zfa8LOhESfsYfq3Dc/061719/

You can easily access this web address using `rclone`. You could set up a new remote in your rclone config to point to `http://sysg1.cs.yale.edu`, but, since you will only be using this once, to get your data, it makes more sense to just specify the remote on the command line. This can be done by passing `rclone` the URL address via the `--http-url` option, and then, after that, telling it what protocol to use by adding `:http:` to the command. Here is what you would use to list the directories available at the sequencing center URL:

```
# here is the command
% rclone lsd --http-url http://sysg1.cs.yale.edu:3010/5lnO9bs3zfa8LOhESfsYfq3Dc/061719/ :h

# and here is the output
          -1 1969-12-31 16:00:00        -1 sjg73_fqs
          -1 1969-12-31 16:00:00        -1 sjg73_supernova_fqs
```

Aha! There are two directories that might hold our sequencing data. I wonder what is in those diretories? The `rclone tree` command is the perfect way to drill down into those diretories and look at their contents:

```
% rclone tree --http-url http://sysg1.cs.yale.edu:3010/5lnO9bs3zfa8LOhESfsYfq3Dc/061719/ :
/
   sjg73_fqs
       AW_F1
           AW_F1_S2_L001_I1_001.fastq.gz
           AW_F1_S2_L001_R1_001.fastq.gz
           AW_F1_S2_L001_R2_001.fastq.gz
       AW_M1
           AW_M1_S3_L001_I1_001.fastq.gz
           AW_M1_S3_L001_R1_001.fastq.gz
           AW_M1_S3_L001_R2_001.fastq.gz
       ESP_A1
           ESP_A1_S1_L001_I1_001.fastq.gz
           ESP_A1_S1_L001_R1_001.fastq.gz
           ESP_A1_S1_L001_R2_001.fastq.gz
   sjg73_supernova_fqs
       AW_F1
           AW_F1_S2_L001_I1_001.fastq.gz
```

```
          AW_F1_S2_L001_R1_001.fastq.gz
          AW_F1_S2_L001_R2_001.fastq.gz
      AW_M1
          AW_M1_S3_L001_I1_001.fastq.gz
          AW_M1_S3_L001_R1_001.fastq.gz
          AW_M1_S3_L001_R2_001.fastq.gz
      ESP_A1
          ESP_A1_S1_L001_I1_001.fastq.gz
          ESP_A1_S1_L001_R1_001.fastq.gz
          ESP_A1_S1_L001_R2_001.fastq.gz

8 directories, 18 files
```

Whoa! That is pretty cool!. From this output we see that there are subdirectories named `AW_F1` and `AW_M1` that hold the files that we want. And, of course, the `ESP_A1` samples must belong to someone else. It would be great if we could just download the files we wanted, excluding the ones in the `ESP_A1` directories. It turns out that there is! `rclone` has an `--exclude` option to exclude paths that match certain patterns (see Section 5.2.4.3, above). We can experiment by giving `rclone copy` the `--dry-run` command to see which files will be transferred. If we don't do any filtering, we see this when we try to dry-run copy the directories to our local directory `Alewife/fastqs`:

```
% rclone copy --dry-run --http-url http://sysg1.cs.yale.edu:3010/5lnO9bs3zfa8LOhESfsYfq3Dc
2019/07/11 10:33:43 NOTICE: sjg73_fqs/ESP_A1/ESP_A1_S1_L001_I1_001.fastq.gz: Not copying a
2019/07/11 10:33:43 NOTICE: sjg73_fqs/ESP_A1/ESP_A1_S1_L001_R1_001.fastq.gz: Not copying a
2019/07/11 10:33:43 NOTICE: sjg73_fqs/ESP_A1/ESP_A1_S1_L001_R2_001.fastq.gz: Not copying a
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_I1_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_R1_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_R2_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_I1_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_R1_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_R2_001.fastq.gz: Not c
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/ESP_A1/ESP_A1_S1_L001_I1_001.fastq.gz: Not
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/ESP_A1/ESP_A1_S1_L001_R1_001.fastq.gz: Not
2019/07/11 10:33:43 NOTICE: sjg73_supernova_fqs/ESP_A1/ESP_A1_S1_L001_R2_001.fastq.gz: Not
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_I1_001.fastq.gz: Not copying as
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_R1_001.fastq.gz: Not copying as
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_R2_001.fastq.gz: Not copying as
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_I1_001.fastq.gz: Not copying as
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_R1_001.fastq.gz: Not copying as
2019/07/11 10:33:43 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_R2_001.fastq.gz: Not copying as
```

Since we do not want to copy the `ESP_A1` files we see if we can exclude them:

```
% rclone copy --exclude */ESP_A1/* --dry-run --http-url http://sysg1.cs.yale.edu:3010/5lnO
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_I1_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_R2_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_F1/AW_F1_S2_L001_R1_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_I1_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_R2_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_fqs/AW_M1/AW_M1_S3_L001_R1_001.fastq.gz: Not copying as
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_I1_001.fastq.gz: Not c
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_R2_001.fastq.gz: Not c
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_F1/AW_F1_S2_L001_R1_001.fastq.gz: Not c
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_I1_001.fastq.gz: Not c
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_R1_001.fastq.gz: Not c
2019/07/11 10:37:22 NOTICE: sjg73_supernova_fqs/AW_M1/AW_M1_S3_L001_R2_001.fastq.gz: Not c
```

Booyah! That gets us just what we want. So, then we remove the `--dry-run` option, and maybe add `-v -P` to give us verbose output and progress information, and copy all of our files:

```
% rclone copy --exclude */ESP_A1/* -v -P  --http-url http://sysg1.cs.yale.edu:3010/5lnO9bs
```

5.3 `tmux`: the terminal multiplexer

Many universities have recently implemented a two-factor authentication requirement for access to their computing resources (like remote servers and clusters). This means that every time you login to a server on campus (using `ssh` for example) you must type your password, and also fiddle with your phone. Such systems preclude the use of public/private key pairs that historically allowed you to access a server from a trusted client (i.e., your own secured laptop) without having to type in a password. As a consequence, today, opening multiple sessions on a server using `ssh` and two-factor authentication requires a ridiculous amount of additional typing and phone-fiddling, and is a huge hassle. But, when working on a remote server it is often very convenient to have multiple separate shells that you are working on and can quickly switch between.

At the same time. When you are working on the shell of a remote machine and your network connection goes down, then, typically the bash session on your remote machine will be forcibly quit, killing any jobs that you might have been in the middle of (however, this is not the case of you submitted those

jobs through a job scheduler like SLURM. Much more on that in the next chapter.). And, finally, in a traditional `ssh` session to a remote machine, when you close your laptop, or put it to sleep, or quit the Terminal application, all of your active bash sessions on the remote machine will get shut down. Consequently, the next time you want to work on that project, after you have logged onto that remote machine you will have to go through the laborious steps of navigating to your desired working directory, starting up any processes that might have gotten killed, and generally getting yourself set up to work again. That is a serious buzz kill!

Fortunately, there is an awesome utility called `tmux`, which is short for "terminal multiplexer" that solves most of the problems we just described. `tmux` is similar in function to a utility called `screen`, but it is easier to use while at the same time being more customizable and configurable (in my opinion). `tmux` is basically your ticket to working *way* more efficiently on remote computers, while at the same time looking (to friends and colleagues, at least) like the full-on, bad-ass Unix user.

In full confession, I didn't actually start using `tmux` until some five years after a speaker at a workshop delivered an incredibly enthusiastic presentation about `tmux` and how much he was in love with it. In somewhat the same fashion that I didn't adopt RStudio shortly after its release, because I had my own R workflows that I had hacked together myself, I thought to myself: "I have public/private key pairs so it is super easy for me to just start another terminal window and login to the server for a new session. Why would I need `tmux`?" I also didn't quite understand how `tmux` worked initially: I thought that I had to run `tmux` simultaneously on my laptop and on the server, and that those two processes would talk to one another. That is not the case! You just have to run `tmux` on the server and all will work fine! The upshot of that confession is that you should *not* be a bozo like me, and you should learn to use `tmux` *right now*! You will thank yourself for it many times over down the road.

5.3.1 An analogy for how `tmux` works

Imagine that the first time you log in to your remote server you also have the option of speaking on the phone to a super efficient IT guy who has a desk in the server room. This dude never takes a break, but sits at his desk 24/7. He probably has mustard stains on his dingy white T-shirt from eating ham sandwiches non-stop while he works super hard. This guy is Tmux.

When you *first* speak to this guy after logging in, you have to preface your commands with `tmux` (as in, "Hey Tmux!"). He is there to help you manage different terminal windows with different bash shells or processes going on in them. In fact, you can think of it this way: you can ask him to set up a terminal (i.e., like a monitor), right there on his desk, and then create a

bunch of windows on that terminal for you—each one with its own bash shell—without having to do a separate login for each one. He has created all those windows, but you still get to use them. It is like he has a miracle-mirroring device that lets you operate the windows that are on the terminal he set up for you on his desk.

When you are done working on all those windows, you can tell Tmux that you want to *detach* from the special terminal he set up for you at the server. In response he says, "Cool!" and shuts down his miracle-mirroring device, so you no longer see those different windows. However, he *does not* shut down the terminal on his desk that he set up for you. That terminal stays on, and any of your processes happening on it keep chugging away…even after you logout from the server entirely, throw the lid down on your laptop, have drinks with your friends at Social, downtown, watch an episode of Parks and Rec, and then get a good night's sleep.

All through the night, Tmux is munching ham sandwiches and keeping an eye on that terminal he set up for you. When you log back onto the server in the morning, you can say "Hey Tmux! I want to attach back to that terminal you set up for me." He says, "No problem!", turns his miracle-mirroring device back on, and in an instant you have all of the windows on that terminal back on your laptop with all the processes still running in them—in all the same working directories—just as you left it all (except that if you were running jobs in those windows, some of those jobs might already be done!).

Not only that, but, further, if you are working on the server when a local thunderstorm fries the motherboard on your laptop, you can get a new laptop, log back into the server and ask Tmux to reconnect you to that terminal and get back to all of those windows and jobs, etc. as if you didn't get zapped. The same goes for the case of a backhoe operator accidentally digging up the fiber optic cable in your yard. Your network connection can go down completely. But, when you get it up and running again, you can say "Hey Tmux! Hook me up!" and he'll say, "No problem!" and reconnect you to all those windows you had open on the server.

Finally, when you are done with all the windows and jobs on the terminal that Tmux set up for you, you can ask him to kill it, and he will shut it down, unplug it, and, straight out of *Office Space*, chuck it out the window. But he will gladly install a new one if you want to start another *session* with him.

That dude is super helpful!

5.3.2 First steps with `tmux`

The first thing you want to do to make sure Tmux is ready to help you is to simply type:

```
% which tmux
```

This should return something like:

```
/usr/bin/tmux
```

If, instead, you get a response like `tmux: Command not found.` then `tmux` is apparently not installed on your remote server, so you will have to install it yourself, or beg your sysadmin to do so (we will cover that in a later chapter). If you are working on the Summit supercomptuer in Colorado or on Hummingbird at UCSC, then `tmux` is installed already. (As of Feb 16, 2020, `tmux` was not installed on the Sedna cluster at the NWFSC, but I will request that it be installed.)

In the analogy, above, we talked about Tmux setting up a terminal in the server room. In `tmux` parlance, such a "terminal" is called a *session*. In order to be able to tell Tmux that you want to reconnect to a session, you will *always* want to name your sessions so you will request a new session with this syntax:

```
% tmux new -s froggies
```

You can think of the `-s` as being short for "session." So it is basically a short way of saying, "Hey Tmux, give me a new session named `froggies`." That creates a new session called `froggies`, and you can imagine we've named it that because we will use it for work on a frog genomics project.

The effect of this is like Tmux firing up a new terminal in his server room, making a window on it for you, starting a new bash shell in that window, *and then giving you control of this new terminal*. In other words, it is sort of like he has opened a new shell window on a terminal for you, and is letting you see and use it on your computer at the same time.

One very cool thing about this is that you just got a new bash shell without having to login with your password and two-factor authentication again. That much is cool in itself, but is only the beginning.

The new window that you get looks a little different. For one thing, it has a section, one line tall, that is green (by default) on the bottom. In our case, on the left side it gives the name of the session (in square brackets) and then the name of the current *window* within that session. On the right side you see the *hostname* (the name of the remote computer you are working on) in quotes, followed by the date and time. The contents in that green band will look something like:

```
[froggies] 0:bash*                                    "login11" 20:02 15-Feb-20
```

This little line of information is the sweet sauce that will let you find your way around all the new windows that `tmux` can spawn for you.

If you are working on a cluster, please pay special attention to the hostname. (In the above case the hostname is `login11`). Many clusters have multiple *login* or *head* nodes, as they are called. The next time you login to the cluster, you might be assigned to a different login node which will have no idea about your `tmux` sessions. If that were the case in this example I would have to use `slogin login11` and authenticate again to get logged into `login11` to reconnect to my `tmux` session, `froggies`. Or, if you were a CSU student and wanted to login specifically to the `login11` node on Summit the next time you logged on you could do `ssh username@colostate.edu@login11.rc.colorado.edu`. Note the specific `login11` in that statement.

Now, imagine that we want to use this *window* in our `froggies` *session*, to look at some frog data we have. Accordingly, we might navigate to the directory where those data live and look at the data with `head` and `less`, etc. That is all great, until we realize that we also want to edit some scripts that we wrote for processing our froggy data. These scripts might be in a directory far removed from the data directory we are currently in, and we don't really want to keep navigating back and forth between those two directories within a single bash shell. Clearly, we would like to have two windows that we could switch between: one for inspecting our data, and the other for editing our scripts.

We are in luck! We can do this with `tmux`. However, now that we are safely working in a *session* that `tmux` started for us, we no longer have to shout "Hey Tmux!" Rather we can just "ring a little bell" to get his attention. In the default `tmux` configuration, you do that by pressing `<cntrl>-b` from anywhere within a `tmux` window. This is easy to remember because it is like a "b" for the "bell" that we ring to get our faithful servant's attention. `<cntrl>-b` is known as the "prefix" sequence that starts all requests to `tmux` from within a session.

The first thing that we are going to do is ask `tmux` to let us assign a more descriptive, name—`data` to be specific—to the current window. We do this with

```
<cntrl>-b ,
```

(That's right! It's a control-b and then a comma. `tmux` likes to get by on a minimum number of keystrokes.) When you do that, the green band at the bottom of the window changes color and tells you that you can rename the

current window. We simply use our keyboard to change the name to "data". That was super easy!

Now, to make a new window with a new bash shell that we can use for writing scripts we do `<cntrl>-b c`. Try it! That gives you a new *window* within the `froggies` session and switches your focus to it. It is as if Tmux (in his mustard-stained shirt) has created a new window on the `froggies` terminal, brought it to the front, and shared it with you. The left side of the green `tmux` status bar at the bottom of the screen now says:

```
[froggies] 0:data- 1:bash*
```

Holy Moly! This is telling you that the `froggies` session has two windows in it: the first numbered 0 and named `data`, and the second numbered 1 and named `bash`. The `-` at the end of `0:data-` is telling you that `data` is the window you were previously focused on, but that now you are currently focused on the window with the `*` after its name: `1:bash*`.

So, the name `bash` is not as informative as it could be. Since we will be using this new window for editing scripts, let's rename it to `edit`. You can do that with `<cntrl>-b ,`. Do it!

OK! Now, if you have been paying attention, you probably realize that `tmux` has given us two windows (with two different bash shells) in this session called `froggies`. Not only that but it has associated a single-digit number with each window. If you are all about keyboard shortcuts, then you probably have already imagined that `tmux` will let you switch between these two windows with `<cntrl>-b` plus a digit (0 or 1 in this case). Play with that. Do `<cntrl>-b 0` and `<cntrl>-b 1` and feel the power!

Now, for fun, imagine that we want to have another window and a bash shell for launching jobs. Make a new window, name it `launch`, and then switch between those three windows.

Finally. When you are done with all that, you tell Tmux to detach from this session by typing:

```
<cntrl>-b d
```

(The `d` is for "detach"). This should kick you back to the shell from which you first shouted "Hey Tmux!" by issuing the `tmux a -t froggies` command. So, you can't see the windows of your `froggies` session any longer, *but do not despair*! Those windows are still on the monitor Tmux set up for you, casting an eerie glow on his mustard stained shirt.

If you want to get back in the driver's seat with all of those windows, you simply need to tell Tmux that you want to be attached again via his miracle-mirroring device. Since we are no longer in a `tmux` window, we don't use our `<cntrl-b>` bell to get Tmux's attention. We have to shout:

```
% tmux attach -t froggies
```

The `-t` flag stands for "target." The `froggies` session is the target of our attach request. Note that if you don't like typing that much, you can shorten this to:

```
% tmux a -t froggies
```

Of course, sometimes, when you log back onto the server, you won't remember the name of the `tmux` session(s) you started. Use this command to list them all:

```
% tmux ls
```

The `ls` here stands for "list-sessions." This can be particularly useful if you actually have multiple sessions. For example, suppose you are a poly-taxa genomicist, with projects not only on a frog species, but also on a fish and a bird species. You might have a separate session for each of those, so that when you issue `tmux ls` the result could look something like:

```
% tmux ls
birdies: 4 windows (created Sun Feb 16 07:23:30 2020) [203x59]
fishies: 2 windows (created Sun Feb 16 07:23:55 2020) [203x59]
froggies: 3 windows (created Sun Feb 16 07:22:36 2020) [203x59]
```

That is enough to remind you of which session you might wish to reattach to.

Finally, if you are all done with a `tmux` session, and you have detached from it, then from your shell prompt (not within a `tmux` session) you can do, for example:

```
tmux kill-session -t  birdies
```

to kill the session. There are other ways to kill sessions while you are in them, but that is not so much needed.

Table 5.1 reviews the minimal set of `tmux` commands just described. Though there is much more that can be done with `tmux`, those commands will get you started.

TABLE 5.1: A bare bones set of commands for using `tmux` The first column says whether the command is given within a `tmux` session rather than at the shell outside of a `tmux` session

Within tmux?	Command	Effect
N	`tmux ls`	List any tmux sessions the server knows about
N	`tmux new -s name`	Create a new tmux session named "name"
N	`tmux attach -t name`	Attach to the existing tmux session "name"
N	`tmux a -t name`	Same as "attach" but shorter.
N	`tmux kill-session -t` name	Kill the tmux session named "name"
Y	`<cntrl>-b ,`	Edit the name of the current window
Y	`<cntrl>-b c`	Create a new window
Y	`<cntrl>-b 3`	Move focus to window 3
Y	`<cntrl>-b &`	Kill current window
Y	`<cntrl>-b d`	Detach from current session
Y	`<cntrl>-l`	Clear screen current window

5.3.3 Further steps with `tmux`

The previous section merely scratched the surface of what is possible with `tmux`.
Indeed, that is the case with this section. But here I just want to leave you with a taste for how to configure `tmux` to your liking, and also with the ability to create different *panes* within a window within a session. You guessed it! A pane is made by splitting a *window* (which is itself a part of a *session*) into two different sections, each one running its own bash shell.

Before we start making panes, we set some configurations that make the establishment of panes more intuitive (by using keystrokes that are easier to remember) and others that make it easier to quickly adjust the size of the panes. So, first, add these lines to `~/.tmux.conf`:

```
# splitting panes
bind \ split-window -h -c '#{pane_current_path}'
bind - split-window -v -c '#{pane_current_path}'
```

```
# easily resize panes with <C-b> + one of j, k, h, l
bind-key j resize-pane -D 10
bind-key k resize-pane -U 10
bind-key h resize-pane -L 10
bind-key l resize-pane -R 10
```

Once you have updated `~/.tmux.conf` you need to reload that configuration file in `tmux`. So, from within a `tmux` session, you do `<cntrl>-b :`. This let's you type a `tmux` command in the lower left (where the cursor has become active). Type `source-file ~/.tmux.conf`

The comments show what each line is intended to do, and you can see that the configuration "language" for `tmux` is relatively unintimidating. In plain language, these configurations are saying that, after this configuration is made active, `<cntrl>-b /` will split a window (or a pane), vertically, in to two panes. (Note that this is easy to remember because on an American keyboard, the \ and the `|`, share a key. The latter looks like a vertical separator, and would thus be a good key stroke to split a screen vertically, but why force ourselves to hit the shift key as well?). Likewise, `<cntrl>-b -` will split a window (or a pane) into two panes.

What do we mean by splitting a window into multiple panes? A picture is worth a thousand words. Figure 5.3 shows a `tmux` window with four panes. The two vertical ones on the left show a yaml file and a shell script being edited in `vim`, and the remaining two house shells for looking at files in two different directories.

This provides almost endless opportunities for customizing the appearance of your terminal workspace on a remote machine for maximum efficiency. Of course, doing so requires you know a few more keystrokes for handling panes. These are summarized in Table 5.2.

TABLE 5.2: Important keystrokes within a `tmux` session for handling panes

Within tmux?	Command	Effect
Y	`<cntrl>-b /`	Split current window/pane vertically into two panes
Y	`<cntrl>-b -`	Split current window/pane horizontally into two panes
Y	`<cntrl>-b arrow`	Use `<cntrl>-b` + an arrow key to move sequentially amongst panes
Y	`<cntrl>-b x`	Kill current the current pane

Within tmux?	Command	Effect
Y	`<cntrl>-b q`	Paint big ID numbers (from 0 up) on the panes for a few seconds. Hitting a number before it disappears moves focus to that pane.
Y	-b [hjkl]	Resize the current pane, h = Left, j = Down, k = Up, l = Right. It takes a while to understand which boundary will move.
Y	-b z	Zoom current pane to full size. `<cntrl>-b z` again restores it to original size.

Now that you have seen all these keystrokes, use `<cntrl>-b \` and `<cntrl>-b -` to split your windows up into a few panes and try them out. It takes a while to get used to it, but once you get the hang of it, it's quite nice.

5.4 `tmux` for Mac users

I put this in an entirely different section, because, if you are comfortable in Mac-world, already, working with tmux by way of the extraordinary Mac application **iTerm2** feels like home and it is a completely different experience than working in `tmux` the way we have, so far.

iTerm2 is a sort of fully customizable and *way* better replacement for the standard Mac Terminal application. It can be downloaded for free from its web page https://www.iterm2.com/. You can donate to the project there as well. If you find that you really like iTerm2, I recommend a donation to support the developers.

There are far too many features in iTerm2 to cover here, but I just want to describe one very important feature: iTerm2 integration with `tmux`. If you have survived the last section, and have gotten comfortable with hitting `<cntrl>-b` and then a series of different letters to effect various changes, then that is a good thing, and will serve you well. However, as you continue your journey with `tmux`, you may have found that you can't scroll up through the screen the way you might be used to when working in Terminal on a Mac. Further, you may have discovered that copying text from the screen, when you finally figured out how to scroll up in it, involves a series of emacs-like keystrokes.

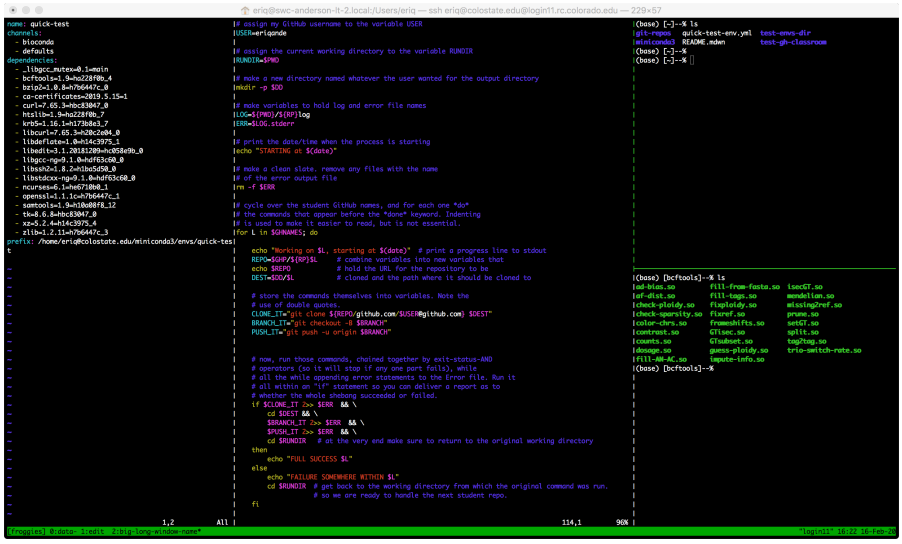


FIGURE 5.3: A tmux window with four panes.

This is fine if you are up for it, but it is understandable that a Mac user might yearn for a more Mac-like experience. Fortunately, the developers of iTerm2 have made your tmux experience much better! They exploit `tmux`'s `-CC` option, which puts `tmux` into "control mode" such that iTerm2 can send its own simple text commands to control `tmux`, rather than the user sending commands prefaced by `<cntrl>-b`. The consequence of this is that iTerm2 has a series of menu options for doing `tmux` actions, and all of these have keyboard shortcuts that seem more natural to a Mac user. You can establish sessions, open new windows (as tabs in iTerm, if desired) and even carve windows up into multiple panels—all from a Mac-style interface that is quite forgiving in case you happen to forget the exact key sequence to do something in `tmux`. Finally, using `tmux` via iTerm2 you get mouse interaction like you expect: you can use the mouse to select different panes and move the dividers between them, and you can scroll back and select text with the mouse, if desired.

On top of that, iTerm2 has great support for creating different profiles that you can assign to different remote servers. These can be customized with different login actions (including storage of remote server passwords in the Apple keychain, so you don't have to type in your long, complex passwords every time you log in to a remote server you can't set a public/private keypair on), and with different color schemes that can help you to keep various windows attached to various remote servers straight.

You can read about it all at the iTerm2 website. I will just add that using iTerm with `tmux` might require a later version of `tmux` than is installed on your remote server or cluster. I recommend that you use Miniconda (see Section 5.5.2) to

install tmux version 2.9 into your base environment with `conda install -c conda-forge tmux=2.9`. Before you do that, be sure to kill all existing tmux sessions on your remote server. Then, you could make a profile named, say `summit-tmux`, that launched with a command that was like this:

```
ssh -t eriq@colostate.edu@login11.rc.colorado.edu "/projects/eriq@colostate.edu/miniconda3
```

But customized to your own account name. What that does when you open a `summit-tmux` session is `ssh` to SUMMIT, and immediately run the command in the double quotes. That command says, "try to attach to a `tmux` session named `summit`. If that fails, then create a new `tmux` session called `summit`." To get the password manager set up for that profile, you need to add to the password manager an account (call it `summit`) and store your summit password in that manager. Then, in your summit-tmux profile, set a trigger (Profile->Advanced->Triggers (Edit)) which looks for the regular expression `^Password:`, does the action "Open Password Manager", and does so instantly. Figure 5.4 is a screen shot of what that setup looks like in my SUMMIT profile:

If you are a Mac user, spend a weekend getting to know iTerm2. It will be time well spent.

5.5 Installing Software on an HPCC

In order to do anything useful on a remote computer or a high-performance computing cluster (called a "cluster" or an "HPCC") you will need to have software programs for analyzing data. As we have seen, a lot of the nuts and bolts of writing command lines uses utilities that are found on every Unix computer. However almost always your bioinformatic analyses will require programs or software that do not come "standard" with Unix. For example, the specialized programs for sequence assembly and alignment will have to be *installed* on the cluster in order for you to be able to use them.

It turns out that installing software on a Unix machine (or cluster) has not always been a particularly easy thing to do for a number or reasons. First, for a long time, Unix software was largely distributed in the form of *source code*: the actual programming code (text) written by the developers that describes the actions that a program takes. Such computer code cannot be run directly, it first must be *compiled* into a *binary* or *executable* program. Doing this can be a challenging process. First, computer code compilation can be very time consuming (if you use R on Linux, and install all your packages from CRAN—which requires compilation—you will know that!). Secondly, vexing errors and

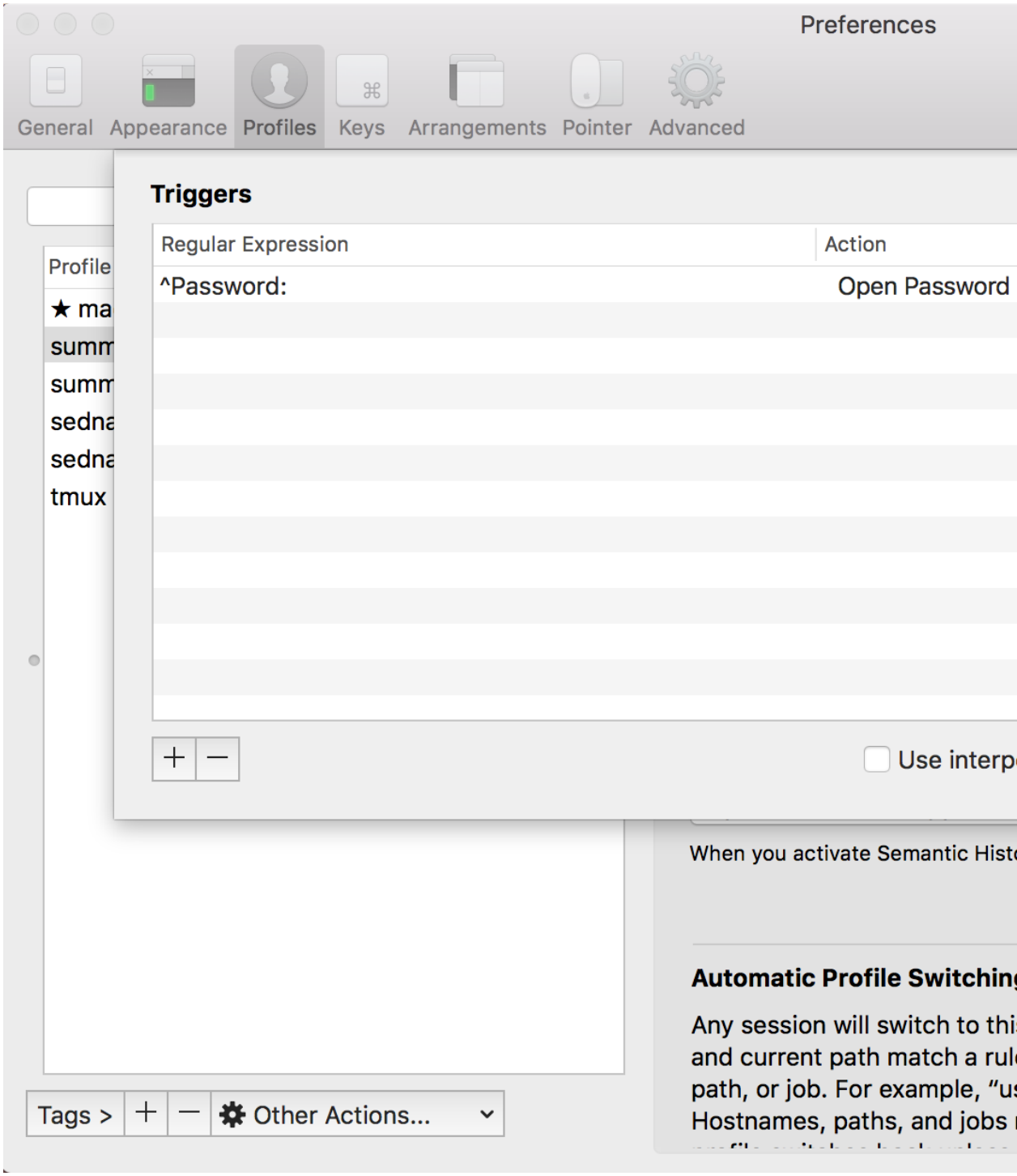


FIGURE 5.4: How to set a trigger to open the password manager in iTerm to be able to pass in your password. This is telling iTerm to open the Password Manager whenever it finds 'Password:' at the beginning of a line in the terminal.

failures can occur when the compiler or the computer architecture is in conflict with the program code. (I have lost entire days trying to solve compiling problems). On top of that, in order to run, most programs do not operate in a standalone fashion; rather, while a program is running, it typically depends on computer code and routines that must be stored in separate *libraries* on your Unix computer. These libraries are known as program *dependencies*. So, installing a program requires not just installing the program itself, but also ensuring that the program's dependencies are installed *and* that the program knows where they are installed. As if that were not enough, the dependencies of some programs can conflict (be incompatible) with the dependencies of other programs, and particular versions of a program might require particular versions of the dependencies. Additionally, some versions of some programs might not work with particular versions (types of chips) of some computer systems. Finally, most systems for installing software that were in place on Unix machines a decade ago required that whoever was installing software have *administrative privileges* on the computer. On an HPCC, none of the typical users have administrative privileges which are, as you might guess, reserved for the system administrators.

For all the reasons above, installing software on an HPCC used to be a harrowing affair: you either had to be fluent in compilers and libraries to do it yourself in your home directory or you had to beg your system administrator. (Though our cluster computing sysadmins at NMFS are wonderful, that is not always the case…see Dilbert[5]). On HPCC's the system administrators have to contend with requests from multiple users for different software and different versions. They solve this (somewhat headachey) problem by installing software into separate "compartments" that allow different software and versions to be maintained on the system without all of it being accessible at once. Doing so, they create *modules* of software. This is discussed in the following section.

Today, however, a large group of motivated people have created a software management system called Miniconda that tries to solve many of the problems encountered in maintaining software on a computer system. First, Miniconda maintains a huge repository of programs that are already *pre-compiled* for a number of different chip architectures, so that programs can usually be installed without the time-consuming compiling process. Second, the repository maintains critical information on the dependencies for each software program, and about conflicts and incompatibilities between different versions of programs, architectures and dependencies. Third, the Miniconda system is built from the ground up to make it easy to maintain separate software *environments* on your system. These different environments have different software programs or different versions of different software programs. Such an approach was originally used so developers could use a single computer to test any new code they had written in a number of different computing environ-

[5] https://www.wiw.org/~chris/archive/dilbert/

ments; however, it has become an incredibly valuable tool for ensuring that your analyses are reproducible: you can give people not just the data and scripts that you used for the analysis, but also the computing/software environment (with all the same software versions) that you used for the analysis. And, finally, all of this can be done with Miniconda without have administrative privileges. Effectively, Miniconda manages all these software programs and dependencies *within your home directory*. Section 5.5.2 provides details about Miniconda and describes how to use it to install bioinformatics software.

5.5.1 Modules

(not written)

This is if your sys admin has made it easy.

The big point I must make here is that these don't work like conda environments. Environments are "all-or-nothing," whereas modules are "cumulative" and can be layered atop one another. (If doing so creates conflicts, then the sysadmins have to figure that out…perhaps this is one reason that modules will often carry older (if not completely antiquated) versions of software.)

5.5.2 Miniconda

We will first walk you through a few steps with Miniconda to install some bioinformatic software into an environment on your cluster. After that we will discuss more about the underlying philosophy of Miniconda, and how it is operating.

5.5.2.1 Installing or updating Miniconda

1. To do installation of software, you probably should not be on SUMMIT's login nodes. They offer "compile nodes" that should be suitable for installing with Miniconda. So, do this:

```
ssh scompile
```

2. If you are on Hummingbird, be sure to get a `bash` shell before doing anything else, by typing `bash` (if you have not already set `bash` as your default shell (see the previous chapter)).
3. First, check if you have Miniconda. Update it if you do, and install it if you don't:

```
# just type conda at the command line:
conda
```

If you see some help information, then you already have Miniconda (or Anaconda) and you should merely update it with:

```
conda update conda
```

If you get an error telling you that your computer does not know about a command, `conda`, then you do not have Miniconda and you must install it. You do that by downloading the Miniconda package with `wget` and then running the Miniconda installer, like this:

```
# start in your home directory and do the following:
mkdir conda_install
cd conda_install/
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
chmod u+x Miniconda3-latest-Linux-x86_64.sh
./Miniconda3-latest-Linux-x86_64.sh
```

That launches the Miniconda installer (it is a shell script). Follow the prompts and agree to the license and to the default install location and to initialize conda. At the end, it tells you to log out of your shell and log back in for changes to take effect. It turns out that it suffices to do `cd ~; source .bash_profile`

Once you complete the above, your command prompt should have been changed to something that looks like:

```
(base) [~]--%
```

The `(base)` is telling you that you are in Miniconda's *base* environment. Typically you want to keep the *base* environment clean of installed software, so we will install software into a new environment.

5.5.2.2 Installing software into a bioinformatics environment

If everything went according to plan above, then we are ready to use Miniconda to install some software for bioinformatics. We will install a few programs that we will use extensively in the next few weeks: `bwa`, `samtools`, and `bcftools`.

We will install these programs into a conda environment that we will name `bioinf` (short for "bioinformatics"). It takes just a single command:

```
conda create -n bioinf -c bioconda bwa samtools bcftools
```

That should only take a few seconds.

To test that we got the programs we must *activate* the `bioinf` environment, and then issue the commands, `bwa`, `samtools`, and `bcftools`. Each of those should spit back some help information. If so, that means they are installed correctly! It looks like this:

```
conda activate bioinf
```

After that you should get a command prompt that starts with `(bioinf)`, telling you that the active conda environment is `bioinf`. Now, try these commands:

```
bwa
samtools
bcftools
```

5.5.2.3 Uninstalling Miniconda and its associated environments

It may become necessary at some point to uninstall Miniconda. One important case of this is if you end up overflowing your home directory with conda-installed software. In this case, unless you have installed numerous, complex environments, the simplest thing to do is to "uninstall" Miniconda, reinstall it in a location with fewer hard-drive space constraints, and then simply recreate the environments you need, as you did originally.

This is actually quite germane to SUMMIT users. The size quota on home directories on SUMMIT is only 2 Gb, so you can easily fill up your home directory by installing a few conda environments. To check how much of the hard drive space allocated to you is in use on SUMMIT, use the `curc-quota` command. (Check the documentation for how to check space on other HPCCs, but note that Hummingbird users get 1 TB on their home directories). Instead of using your home directory to house your Miniconda software, on SUMMIT you can put it in your `projects` storage area. Each user gets more storage (like 250 Gb) in a directory called `/projects/username` where `username` is replaced by your SUMMIT username, for example: `/projects/eriq@colostate.edu`

To "uninstall" Miniconda, you first must delete the `miniconda3` directory in your home directory (if that is where it got installed to). This can take a while. It is done with:

```
rm -rf ~/miniconda3
```

Then you have to delete the lines between `# >>>` and `# <<<`, wherever they occur in your `~/.bashrc` and `~/bash_profile` files, i.e., you will have to remove all of the lines that look something like thius:

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/eriq/miniconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/Users/eriq/miniconda3/etc/profile.d/conda.sh" ]; then
        . "/Users/eriq/miniconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/eriq/miniconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

After all those conda lines are removed from your `~/.bashrc` and `~/bash_profile`, logging out and logging back in, you should be free from conda and ready to reinstall it in a different location.

To reinstall miniconda in a different location, just follow the installation instructions above, but when you are running the `./Miniconda3-latest-Linux-x86_64.sh` script, instead of choosing the default install location, use a location in your project directory. For example, for me, that is: `/projects/eriq@colostate.edu/miniconda3`.

Then, recreate the `bioinf` environment described above.

If you are having fun making environments and you think that you might like to use R on the cluster, then you might want to make an environment with some bioinformatics software that also has the latest version of R on miniconda installed. At the time of writing that was R 3.6.1. So, do:

```
conda create -n binfr -c bioconda bwa samtools bcftools r-base=3.6.1 r-essentials
```

That makes an environment called `binfr` (which turns out to also be **way** easier to type that `bioinfr`). The `r-essentials` in the above command line is the name for a collection of 200 commonly used R packages (including the `tidyverse`). This procedure takes a little while, but it is still far less painful than using the version of R that is installed on SUMMIT with the

`modules` packages, and then trying to build the tidyverse from source with `install.packages()`.

5.5.2.4 What is Miniconda doing?

This is a good question. We won't go deeply into the specifics, but will skim the surface of a few topics that can help you understand what Miniconda is doing.

First, Miniconda is downloading programs and their dependencies into the `miniconda3` directory. Based on the lists of dependencies and conflicts for each program that is being installed, it makes a sort of "equation," which it can "solve" to find the versions of different programs and libraries that can be installed and which should "play nicely with one another (and with your specific computer architecture." While it is solving this "equation" it is doing so while also doing its best to optimize features of the programs (like using the latest versions, if possible). Solving this "equation" is an example of a Boolean Satisfiability problem, which is a known class of difficult (time-consuming) problems. If you are requesting a lot of programs, and especially if you do not constrain your request (by demanding a certain version of the program) then "solving" the request may take a long time. However, when installing just a few bioinformatics programs it is unlikely to ever take too terribly long.

Once miniconda has decided on which versions of which programs and dependencies to install, it downloads them and then places them into the requested environment (or the active environment if no environment is specifically requested). If a program is installed into an environment, then you can access that program by activating the environment (i.e. `conda activate bioinf`). Importantly, if you don't activate the environment, you won't be able to use the programs installed there. We will see later in writing bioinformatic scripts, you will always have to explicitly activate a desired conda environment when you run a script on a compute node through the job scheduler.

The way that Miniconda delivers programs in an environment is by storing all the programs in a special environment directory (within the `miniconda3/envs` directory), and then, when the environment is activated, the main thing that is happening is that `conda` is manipulating your PATH variable to include directories within the environment's directory within the `miniconda3/envs` directory. An easy way to see this is simply by inspecting your path variable while in different environments. Here we compare the PATH variable in the `base` environment, versus in the `bioinf` environment, versus in the `binfr` environment:

```
(base) [~]--% echo $PATH
/projects/eriq@colostate.edu/miniconda3/bin:/projects/eriq@colostate.edu/miniconda3/condab
(base) [~]--% conda activate bioinf
(bioinf) [~]--% echo $PATH
/projects/eriq@colostate.edu/miniconda3/envs/bioinf/bin:/projects/eriq@colostate.edu/minic
(bioinf) [~]--% conda activate binfr
(binfr) [~]--% echo $PATH
/projects/eriq@colostate.edu/miniconda3/envs/binfr/bin:/projects/eriq@colostate.edu/minico
```

(To be sure, `miniconda` can change a few more things than just your PATH variable when you activate an environment, but for the typical user, the changes to PATH are most important.)

5.5.2.5 What programs are available on Minconda?

There are quite a few programs for multiple platforms. If you are wondering whether a particular program is available from Miniconda, the easiest first step is to Google it. For example, search for `miniconda bowtie`.

You can also search from the command line using `conda search`. Note that most bioinformatics programs you will be interested in are available on a conda *channel* called `bioconda`. You probably saw the `-c bioconda` option applied to the `conda create` commands above. That options tells conda to search the Bioconda channel for programs and packages.

Here, you can try searching for a couple of packages that you might end up using to analyze genomic data:

```
conda search -c bioconda plink

# and next:

conda search -c bioconda angsd
```

5.5.2.6 Can I add more programs to an environment?

This is a worthwhile question. Imagine that we have been happily working in our `bioinf` conda environment for a few months. We have finished all our tasks with `bwa`, `samtools`, and `bcftools`, but perhaps now we want to analyze some of the data with `angsd` or `plink`. Can we add those programs to our `bioinf` environment? The short answer is "Yes!". The steps are easy.

1. Activate the environment you wish to add the programs to (i.e. `conda activate bioinf` for example).
2. Then use `conda install`. For example to install specific versions of `plink` and `angsd` that we saw above while searching for those packages we might do:

```
conda install -c bioconda plink=1.90b6.12 angsd=0.931
```

Now, the longer answer is "Yes, but…" The big "but" there occurs because if different programs require the same dependencies, but rely on different versions of the dependencies, installing programs over different commands can cause miniconda to not identify some incompatibilities between program dependencies. A germane example occurs if you first install `samtools` into an environment, and then, after that, you install `bcftools`, like this:

```
conda create -n samtools-first        # create an empty environment
conda activate samtools-first         # activate the environment
conda install -c bioconda samtools    # install samtools
conda install -c bioconda bcftools    # install bcftools
bcftools                              # try running bcftools
```

When you try running the last line, `bcftools` barfs on you like so:

```
bcftools: error while loading shared libraries: libcrypto.so.1.0.0: cannot open shared obj
```

So, often, installing extra programs does not create problems, but it can. If you find yourself battling errors from conda-installed programs, see if you can correct that by creating a new environment and installing all the programs you want at the same time, in one fell swoop, using `conda create`, as in:

```
conda create -n binfr -c bioconda bwa samtools bcftools r-base=3.6.1 r-essentials
```

5.5.2.7 Exporting environments

In our introduction to Miniconda, we mentioned that it is a great boon to reproducibility. Clearly, your analyses will be more reproducible if it is easier for others to install software to repeat your analyses. However, Miniconda takes that one step further, allowing you to generate a list of the specific versions of all software and dependencies in a conda environment. This list is a complete record of your environment, and, supplied to conda, it is a specification of exactly how to recreate that environment.

The process of creating such a list is called *exporting* the conda environment. Here we demonstrate its use by exporting the `bioinf` environment from SUMMIT to a simple text file. Then we use that text file to recreate the environment on my laptop.

```
# on summit:
conda activate bioinf          # activate the environment
conda env export               # export the environment
```

The last command above just sends the exported environment to stdout, looking like this:

```
name: bioinf
channels:
  - bioconda
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - bcftools=1.9=ha228f0b_4
  - bwa=0.7.17=hed695b0_7
  - bzip2=1.0.8=h7b6447c_0
  - ca-certificates=2020.1.1=0
  - curl=7.68.0=hbc83047_0
  - htslib=1.9=ha228f0b_7
  - krb5=1.17.1=h173b8e3_0
  - libcurl=7.68.0=h20c2e04_0
  - libdeflate=1.0=h14c3975_1
  - libedit=3.1.20181209=hc058e9b_0
  - libgcc-ng=9.1.0=hdf63c60_0
  - libssh2=1.8.2=h1ba5d50_0
  - libstdcxx-ng=9.1.0=hdf63c60_0
  - ncurses=6.1=he6710b0_1
  - openssl=1.1.1d=h7b6447c_4
  - perl=5.26.2=h14c3975_0
  - samtools=1.9=h10a08f8_12
  - tk=8.6.8=hbc83047_0
  - xz=5.2.4=h14c3975_4
  - zlib=1.2.11=h7b6447c_3
prefix: /projects/eriq@colostate.edu/miniconda3/envs/bioinf
```

The format of this information is YAML (Yet Another Markup Language), (which we saw in the headers of RMarkdown documents, too).

If we stored that output in a file:

```
conda env export > bioinf.yml
```

And then copied that file to another computer, then we can recreate the environment on that other computer with:

```
conda env create -f bioinf.yml
```

That should work fine if the new computer is of the same architecture (i.e., both are Linux computers, or both are Macs). However, the specific build numbers referenced in the YAML (i.e. things like the `h7b6447c_3` part of the program name) can create problems when installing on other architectures. In that case, we must export without the build names:

```
conda env export --no-builds > bioinf.yml
```

Even that might fail if the dependencies differ on different architectures, in which case you can export just the list of the actual programs that you reqested be installed, by using the `--from-history` option. For example:

```
% conda env export --from-history
name: bioinf
channels:
  - defaults
dependencies:
  - bwa
  - bcftools
  - samtools
prefix: /projects/eriq@colostate.edu/miniconda3/envs/bioinf
```

Though, even that fails, cuz it doesn't list bioconda in there.

5.6 vim: it's time to get serious with text editing

Introduce newbs to the `vimtutor`.

5.6.1 Using neovim and Nvim-R and tmux to use R well on the cluster

These are currently just notes to myself.

On Summit you can follow the directions install Neovim and Nvim-R etc, found at section 2 of https://gist.github.com/tgirke/7a7c197b443243937f68c422e5471899#ucrhpcc. You can just do 2.1 to 2.6. 2.7 is the routine for user accounts. You don't need to install Tmux.

You need to get an interactive session on a compute node and then

```
module load R/3.5.0
module load intel
module load mkl
```

The last two are needed to get a random number to start up client through R. It is amazing to me that they call a specific Intel library to do that, and apparently loading the R module alone doesn't get you that.

Uncomment the lines:

```
let R_in_buffer = 0
let R_tmux_split = 1
```

in your `~/.config/nvim/init.vim`. Wait! You don't want to do that, necessarily, because tmux with NVim-R is no longer supported (Neovim now has native terminal splitting support.)

6

High Performance Computing Clusters (HPCC's)

One interesting aspect of modern next generation sequencing data is simply its sheer size: it is not uncommon to receive a half or a full terabyte of data from a sequencing center. It is impractical to store this much data on your laptop, let alone analyze it there. Crunching through such a quantity of data on a single computer or server could take a very long time. Instead, you will likely break up the analysis of such data into a number of smaller jobs, and send them off to run on an assortment of different computers in a High Performance Computing Cluster (HPCC).

Thus, even if you have immersed yourself in bioinformatic data file formats and honed your skills at shell programming and accessing remote computers, sequence data analysis remains such a formidable foe that there is still one last key area of computing in which you must be fluent, in order to comfortably do bioinformatics: you must understand how to submit and manage jobs sent to an HPCC.

My first experience with HPCC's occurred when I started analyzing high-throughput sequencer output. I had over 15 years experience in shell programming at that time, and I was given some example analysis scripts to emulate, but I still found it took several weeks before I was moderately comfortable in an HPCC environment. Most HPCC's have some sort of tutorial web pages that provide a little bit of background on cluster computing, but I didn't find the ones available to me, at the time, to be particularly helpful.

The goal of this chapter is to provide the sort of background I wish that I had when I started doing cluster computing for bioinformatics. I will not be providing a comprehensive overview of parallel computation. For example, we will not focus at all upon the rich tradition of parallel computing applications through "message passing" interfaces which can maintain a synchronized analysis from a single program executing on multiple computers at the same time. Rather, we will focus on the manner in which most bioinformatic problems can be broken down into a series of smaller jobs, each of which can be run, independently, on its own processor without the need for maintaining synchrony between multiple processes.

We start with an overview of what an HPCC consists of, defining a few impor-

tant terms. Then we provide some background on the fundamental problem of cluster computing: namely that a lot of people want to use the computing power of the cluster, but it needs to be allocated to users in an equitable fashion. An understanding of this forms the basis for our discussion of *job scheduling* and the methods you must use to tell the *job scheduler* what resources you will need, so that those resources will eventually be allocated to you. We will cover a job scheduler called SLURM, which stands for the Simple Linux Utility for Resource Management. It is the scheduler used on the Summit supercomputer in Boulder and the Hummingbird and Sedna clusters deployed at UCSC and the NWFSC, respectively.

Interspersed with our discussion of SLURM, we will cover methods for installing software to use for your analyses.

6.1 An oversimplified, but useful, view of a computing cluster

At its simplest, an HPCC can be thought of as a whole lot of computers that are all put in a room somewhere for the purposes of doing a lot of computations. Most of these computers do not possess all the elements that typically come to mind when you think of a "computer." For example, none of them are attached to monitors—each computer resembles just the "box" part of a desktop computer. Each of these "boxes" is called a *node*. The node is the unit in a cluster that corresponds most closely to what you think of as a "computer." As is typical of most computers today, each of these nodes has some amount of Random Access Memory (RAM) that is *only accessible by the node itself*. RAM is the space in memory that is used for active computation and calculation.

Each of these nodes might also have a hard drive used for operating system software. The bulk of the hard drive space that each node can access, however, is in the form of a large array of hard disks that are connected to *all* of the nodes by an interface that allows data to be transferred back and forth between each node and the hard-drive array at speeds that would be expected of an internal hard drive (i.e., this array of hard drives is not just plugged in with a USB cable). Memory on hard drives is used for storing data and the results of calculations, but is not used for active calculations the way RAM is used. In order for calculations to be done on data that are on the hard drive array, it must first be read into a node's RAM. After the calculations are done, the results are typically written back out onto the hard drive array.

On a typical cluster, there are usually several different "portions" of the hard

drive array attached to every *node*. One part of the array holds *home directories* of the different users. Each user typically has a limited amount of storage in their home directory, but the data in these home directories is usually safe or protected, meaning you can put a file there and expect that it will be there next week, month, or year, etc. It is also likely that the home directories are backed up (but check the docs for your cluster to confirm this!). Since the space in home directories is limited, you typically will not put large data sets in your home directory for long-term storage, but you will store scripts and programs, and such items as cloned GitHub repositories there. Another type of storage that exists on the hard drive array is called *persistent long-term storage*. This type of storage is purchased for use by research groups to store large quantities of data on the cluster for long periods of time. As discussed in the last chapter, the rise of cloud-based storage solutions, like Google Drive, offering unlimited storage to institutional users, makes persistent long-term storage less important (and less cost-effective) for many research groups. Finally, the third type of storage in the hard drive array is called *scratch storage*. There are usually fairly light limits (if any at all) to how much data can be placed in scratch storage, but there will typically be Draconian *time limits* placed on your scratch storage. For example, on the Hoffman2 cluster at UCLA, you are granted 2 Tb of `scratch` storage, but any files that have sat unmodified in `scratch` for more than 14 days will be deleted (and if space is tight, the system administrators may delete things from `scratch` in far fewer then 14 days.) Check your local cluster documentation for information about time and space limits on `scratch`.

On many clusters, scratch space is also configured to be very fast for input and output (for example, on many systems, the scratch storage will be composed of solid state drives rather than spinning hard disks). On jobs that require that a lot of data be accessed from the drive or written to it (this includes most operations on BAM files), significant decreases in overall running time can be seen by using fast storage. Finally, scratch space exists on a cluster expressly as *the place* to put data and outputs on the hard drive *when running jobs*. For all these reasons, when you run jobs, you will always want to read and write data from and to `scratch`. We will talk more about the specifics of doing so, but for now, you should be developing a generic picture of your cluster computing workflow that looks like:

1. Download big data files from the cloud to `scratch`
2. Run analyses on those data files, writing output back to `scratch`.
3. When done, copy, to the cloud, any products that you wish to keep.
4. Remove data and outputs left on `scratch`.

As is the case with virtually all modern desktop or laptop computers, within each node, there are multiple (typically between 16 and 48) *cores*, which are the computer chip units that actually do the computations within a node. A

serial job is one that just runs on a single core within a *node*, while a *parallel job* might run on multiple cores, at the same time, within a single node. In such a parallel job, each core has access to the same data within the node's RAM (a "shared-memory," parallel job). The *core* is the fundamental unit of computing machinery that gets allocated to perform jobs in an HPCC.

Most of the nodes in a cluster are there to hold the cores which are the computational workhorses, slogging through calculations for the HPCC's myriad users. However, some nodes are more appropriate to certain types of computations than others (for example, some might have lots of memory for doing genome assembly, while others will have big, hurkin', graphical processing units to be used for GPU calculations). Or, some nodes might be available preferentially for different users than for others. For these reasons, nodes are grouped into different collections. Depending on the system you are using, these collections of different nodes are called, either, *partitions* or *queues*. The world of SLURM uses the term *partitions* for these collections, and we will adopt that language as well, using *queue* to refer to the line of jobs that are waiting to start on an HPCC; however we warn the reader that other job schedulers (like the Univa Grid Engine) use the term "queues" to refer to collections of nodes.

On every cluster, however, there will be one to several nodes that are reserved not for doing computation, but for allowing users to access the cluster. These are called the *login* nodes or the *head* nodes. These nodes are *solely* for logging in, light editing of scripts, minor manipulation of directories, and scheduling and managing jobs. They are absolutely *not* for doing major computations. For example, you should never login to the head node and immediately start using it, in an interactive `bash` session to, say, sort BAM files or run `bwa mem` to do alignments. Running commands that require a lot of computation, or a lot of input and output from the disk, on the login nodes is an egregious *faux pas* of cluster computing. Doing so can negatively impact the ability of other users to login or otherwise get their work done, and it might lead the system administrators to disable your account. Therefore, never do it! All of your hardcore computation on a cluster *has* to be done on a *compute node*. We will show how to do that shortly, but first we will talk about why.

6.2 Cluster computing and the job scheduler

When you do work, or stream a video, or surf the web on your laptop computer, there are numerous different computer processes running, to make sure that your computer keeps working and doing what it is supposed to be doing. In the case of your laptop, the operating system, itself, orchestrates all these

different processes, making sure that each one is given some compute time on your laptop's processors in order to get its work done. Your laptop's operating system has a fair bit of flexibility in how it allocates resources to these different processes: it has multiple cores to assign different processes to, *and* it allows multiple processes to run on a single core, alternating between these different processes over different *cycles* of the central processing unit. Things work differently on a shared resource like an HPCC. The main, interesting problem of cluster computing is basically this: lots of people want to use the cluster to run big jobs, but the cluster does not run like a single computer.

The cluster is not happy to give lots of different jobs from lots of different users a chance to all run on the same core, sharing time by dividing up cycles. When a user wants to use the computational resources of an HPCC, she cannot just start a job and be confident that it will launch immediately and be granted at least a few CPU cycles every now and again. Rather, on an HPCC, every *job* that is submitted by a user will be assigned to a dedicated core (or several cores, if requested, and granted) with a dedicated amount of memory. If a core is not available for use, the job "gets in line" (into the "queue", so to speak) where it sits and waits (doing nothing) until a core (with sufficient associated resources, like RAM memory) becomes available. When such a core becomes available in the cluster, the job gets launched on it. All of this is orchestrated by the job scheduler, of which SLURM is an example.

In this computing model, a job, once it is launched, ties up the core and the memory that has been allocated to it until the job is finished. While that job is running, no one else's jobs or processes can run on the core or share the RAM memory that was allocated to the job. For this reason, the job scheduler, needs to know, *ahead of time* how long each job might run and what resources will be required during that time. A simple contrived example illustrates things easily: imagine that Joe and Cheryl each have 1000 separate jobs to run. Each of Cheryl's jobs involves running a machine-learning algorithm to identify seabirds in high-resolution, aerial images of the ocean, and takes only about 20 minutes running on a single core. Each of Joe's jobs, on the other hand, involves mapping billions of sequencing reads, a task which requires about 36 hours when run on a single core. If their cluster has only 640 cores, and Joe submits his jobs first, then, if the job scheduler were naive, it might put all of his jobs in line first, requiring some 50 or 60 hours before the first of Cheryl's jobs even runs. This would be a huge buzz kill for Cheryl. However, if Cheryl and Joe both have to provide estimates to the scheduler of how long their jobs will run, the scheduler can make more equitable decisions, starting a few of Joe's jobs, but retaining many more cores for Cheryl's jobs, each of which runs much faster.

Thus, when you want to run any jobs on a cluster, you must provide an estimate of the resources that the job will require. The three main axes upon which these resources are measured are:

1. The number of cores the job will require.
2. The maximum amount of RAM (memory) the job will require.
3. The amount of time for which the job will run.

Requests for large amounts of resources for long periods of time generally take longer to start. There are two main reasons for this: either 1) the scheduler does not want to launch too many long-duration, high-memory jobs because it anticipates other users will want to use resources down the road and no single user should tie up the compute resources for too long; or 2) there are so many jobs running on myriad nodes and cores, that only infrequently do nodes with sufficient numbers of cores and RAM come available to start the new jobs.

The second reason is a particular bane of new cluster users who unwittingly request more resources than actually exist (i.e. 52 cores, when no single node has more than 32; or 50 Gb of RAM when no single node has more than 48 Gb). Unfortunately (or, perhaps comically, if you have a sick sense of humor), most job schedulers will not notify you of this sort of transgression. Rather, your job will just sit in line waiting to be launched, but it never will be, because sufficient resources never become available!

It is worth noting that regardless of whether reason 1 or reason 2 is the dominant cause influencing how long it takes to start a job, asking for fewer resources for less time will generally allow your jobs to start faster. Particularly because of reason #2, however, breaking your jobs down (if possible) into small chunks that will run relatively quickly on a single core with low RAM needs can render many more opportunities for your jobs to start, letting you tap into resources that are not often fully utilized in a cluster. Since I started working on large cluster in which it took a long time to start a job that required all or most of the cores on a single node, but in which there were many nodes harboring a few cores that were not being used, I tend to endorse this approach…

Since the requested resources for a job play such a large role in wait times for jobs to start, you might wonder why people don't intentionally underestimate the resources they request for their jobs. The answer is simple: the job scheduler is a highly efficient and completely dispassionate manager. If you requested 2 hours for your job, but your job has not finished in that amount of time, the job scheduler will waste no time hemming and hawing or having an emotional struggle with itself about whether it should stop your job. No, at 2 hours and 0.2 seconds it WILL kill your job, regardless of whether it is just about to finish, or not. Similarly, if you requested 4 Gb or RAM, but five hours into your job, the program you are running ends up using 5 Gb or RAM to store a large chunk of data, your job WILL be killed, immediately.

Thus, it is best to be able to accurately estimate the time and resources a job will require. You always want to request more time and resources than your

job will actually need, but not too much more. A large part of getting good at computing in a shared cluster resource is gaining experience in predicting how long different jobs will run, and how much RAM they will require. Later we will describe how the records of your jobs, stored by the job scheduler, can be accessed and organized to aid in predicting the resource demand of future jobs.

6.3 Learning about the resources on your HPCC

Most computing clusters have a web page that describes the configuration of the system and the resources available on it. However, you can use various SLURM commands to learn about those resources as well, and also to gain information about which of the resources are in use and how many users are waiting to use them.

Requests for resources, and for *information* about the computing cluster, are made to the job scheduler using a few basic commands. As said, we will focus on the commands available in a SLURM-based system. In a later section, (after a discussion of installing software that you might need on your cluster) we will more fully cover the commands used to *launch*, *schedule*, and manage jobs. Here we will first explore the SLURM commands that you can use to "get to know" your cluster.

All SLURM commands begin with an `s`, and all SLURM systems support the `sinfo` command that gives you information about the cluster's nodes and their status (whether they are currently running jobs or not.) On your cluster, `man sinfo` will tell you about this command. On the `Sedna` cluster, which just got installed and therefore does not have many users, we see:

```
% sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
nodes*       up   infinite     28   idle node[01-28]
himem        up   infinite      1  alloc himem01
himem        up   infinite      2   idle himem[02-03]
```

which tells us that there are two *partitions* (collection of nodes) named `nodes` and `himem`. The `himem` partition has one node which is currently allocated to a job (`STATE` = `alloc`), and two more that are not currently allocated to jobs. The `himem` partition holds machines with lots of RAM memory for tasks like sequence assembly (and, when I ran that command, one of the nodes in `himem` was busy assembling the genome of some interesting sea creature [need to ask

Krista again what the hell it was...]. It also has 28 compute nodes in the `nodes` partition that are free. This is a very small cluster.

If you do the same command on SUMMIT you get many more lines of output. There are many more different partitions, and there is a lot of information about how many nodes are in each:

```
% sinfo
PARTITION        AVAIL  TIMELIMIT  NODES  STATE NODELIST
shas*               up 1-00:00:00      2 drain* shas[0136-0137]
shas*               up 1-00:00:00      2  down* shas[0404,0506]
shas*               up 1-00:00:00      1  drain shas0101
shas*               up 1-00:00:00      3   resv shas[0102,0521,0853]
shas*               up 1-00:00:00     74    mix shas[0125,0130,0133,0138,0141,0149,0156,01
shas*               up 1-00:00:00    359  alloc shas[0103-0124,0126-0129,0131-0132,0134-01
shas*               up 1-00:00:00     11   idle shas[0226,0244-0245,0403,0441,0557-0559,06
shas-testing        up   infinite      2 drain* shas[0136-0137]
shas-testing        up   infinite      2  down* shas[0404,0506]
shas-testing        up   infinite      1  drain shas0101
shas-testing        up   infinite      3   resv shas[0102,0521,0853]
shas-testing        up   infinite     74    mix shas[0125,0130,0133,0138,0141,0149,0156,01
shas-testing        up   infinite    359  alloc shas[0103-0124,0126-0129,0131-0132,0134-01
shas-testing        up   infinite     11   idle shas[0226,0244-0245,0403,0441,0557-0559,06
shas-interactive    up   infinite      2 drain* shas[0136-0137]
shas-interactive    up   infinite      2  down* shas[0404,0506]
shas-interactive    up   infinite      1  drain shas0101
shas-interactive    up   infinite      3   resv shas[0102,0521,0853]
shas-interactive    up   infinite     74    mix shas[0125,0130,0133,0138,0141,0149,0156,01
shas-interactive    up   infinite    359  alloc shas[0103-0124,0126-0129,0131-0132,0134-01
shas-interactive    up   infinite     11   idle shas[0226,0244-0245,0403,0441,0557-0559,06
sgpu                up 1-00:00:00      1   resv sgpu0501
sgpu                up 1-00:00:00      1  alloc sgpu0502
sgpu                up 1-00:00:00      9   idle sgpu[0101-0102,0201-0202,0301-0302,0401-04
sgpu-testing        up   infinite      1   resv sgpu0501
sgpu-testing        up   infinite      1  alloc sgpu0502
sgpu-testing        up   infinite      9   idle sgpu[0101-0102,0201-0202,0301-0302,0401-04
sknl                up 1-00:00:00      1  drain sknl0710
sknl                up 1-00:00:00      1   resv sknl0706
sknl                up 1-00:00:00     18  alloc sknl[0701-0705,0707-0709,0711-0720]
sknl-testing        up   infinite      1  drain sknl0710
sknl-testing        up   infinite      1   resv sknl0706
sknl-testing        up   infinite     18  alloc sknl[0701-0705,0707-0709,0711-0720]
smem                up 7-00:00:00      1   drng smem0201
smem                up 7-00:00:00      4  alloc smem[0101,0301,0401,0501]
ssky                up 1-00:00:00      1   drng ssky0944
```

```
ssky                up 1-00:00:00      1    mix ssky0952
ssky                up 1-00:00:00      3  alloc ssky[0942-0943,0951]
ssky-preemptable    up 1-00:00:00      1   drng ssky0944
ssky-preemptable    up 1-00:00:00      1    mix ssky0952
ssky-preemptable    up 1-00:00:00      9  alloc ssky[0933-0934,0937-0940,0942-0943,0951]
ssky-preemptable    up 1-00:00:00      9   idle ssky[0935-0936,0941,0945-0950]
ssky-ucb-aos        up 7-00:00:00      6  alloc ssky[0933-0934,0937-0940]
ssky-ucb-aos        up 7-00:00:00      3   idle ssky[0935-0936,0941]
ssky-csu-mbp        up 7-00:00:00      1   idle ssky0945
ssky-csu-asb        up 7-00:00:00      1   idle ssky0946
ssky-csu-rsp        up 7-00:00:00      4   idle ssky[0947-0950]
```

Yikes! That is a lot of info. The numbers that you see on the ends of the lines there are node numbers.

If you wanted to see information about each node in the `shas` partition, you could print long information for each node like this:

```
% sinfo -l -N
```

On Summit, that creates almost 1500 lines of output. Some nodes are listed several times because they belong to different partitions. To look at results for just the standard compute parition (`shas`: 380 nodes with Intel Xeon Haswell processors) you can use

```
% sinfo -l -n -p shas
```

Try that command. (Or try a similar command with an appropriate partition name on your own cluster.) If you want to see explicitly how many cores are available vs allocated on each node, how much total memory each node has, and how much of that total memory is free, in that partition you can do:

```
% sinfo -N -p shas -O nodelist,cpusstate,memory,allocmem,freemem
```

The top part of the output from that command on SUMMIT looks like:

```
NODELIST            CPUS(A/I/O/T)       MEMORY              ALLOCMEM            FREE_MEM
shas0101            0/0/24/24           116368              0                   125156
shas0102            0/24/0/24           116368              0                   112325
shas0103            24/0/0/24           116368              116352              87228
shas0104            24/0/0/24           116368              116352              80769
```

This says `shas0101` has 24 CPUs that are *Out* (not functional at this point). `shas0102`, on the other hand, has 24 CPUs that are *Idle*, while `shas0103` has

24 CPUs that are allocated, and so forth. (In our parlance, here, CPU is being used to mean "core"). All of the nodes have 116 Gb of memory total. Most of them have about that much memory allocated to the jobs they are running.

We can throw down some `awk` to count the total number of available cores (and the total number of all the cores):

```
% sinfo -N -p shas -O  nodelist,cpusstate | awk -F"/" '{avail+=$2; tots+=$4} END {print "O
Out of 10848 cores, there are 331 available
```

That could explain why it can be hard to get time on the supercomputer: at this time, only about 3% of the cores in the system are idle, waiting for jobs to go on them.

If you want to see how many jobs are in line, waiting to be launched, you can use the `squeue` command. Simply issuing the command `squeue` will give a (typically very long) list of all jobs that are either currently running or are in the queue, waiting to be launched. This is worth doing in order to see how many different people are using (or waiting to use) the resources.

If we run the command on Sedna, we see that (like we saw before) only one job is currently running:

```
% squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               160     himem MaSuRCA_   ggoetz  R 34-15:56:16      1 himem01
```

Holy cow! Looking at the TIME column you can see that the genome assembly job has been running for over 34 days! (The time format is DAYS-Hours:Minutes:Seconds).

The output can be filtered to just PENDING or RUNNING jobs using the `-t` option. So,

```
squeue -t PENDING
```

lists all jobs waiting to be launched. If you wanted to see jobs that a particular user (like yourself) has running or pending, you can use the `-u` option. For example,

```
squeue -u eriq@colostate.edu
```

In fact that is such a worthwhile command that it is worth creating an alias for it by adding a line such as the following to your `~/.bashrc`:

```
alias myjobs='squeue -u eriq@colostate.edu'
```

Then, typing `myjobs` shows all of the jobs that you have running, or waiting to be launched on the cluster.

6.4 Getting compute resources allocated to your jobs on an HPCC

6.4.1 Interactive sessions

Although most of your serious computing on the cluster will occur in *batch* jobs that get submitted to the scheduler and run without any further interaction from you, the user, we will begin our foray into the SLURM job scheduling commands by getting an *interactive shell* on a compute node. "Why?" you might ask? Well, before you launch any newly-written script as a *batch* job on the cluster, you really should run through the code, line by line, inside that script and ensure that things work as they should. Since your script likely does some heavy computing, you can't do this on the head nodes (i.e., login nodes) of the cluster.

The system administrators of every different HPCC seem to have their own preferred (or required) way of requesting an interactive shell on a compute node. Before you start doing work on an HPCC, you should always read through the documents (usually available on a website associated with the HPCC) to learn the accepted way of doing certain tasks (like requesting an interactive shell). The following is a short survey of three different ways I have seen for requesting an interactive session. None of these methods is universally applicable to all HPCCs. (Again, read the documents for your own HPCC.)

One way to get an interactive shell on a compute node of a cluster is to request it with:

```
srun --pty /bin/bash
```

This tells slurm to run `bash` (which is typically found at `/bin/bash` on all Unix system) within a pseodo-terminal (the `--pty` part). If you run this command you might be given a bash shell on a compute node. This works on the Sedna cluster, but not on all others.

On the SUMMIT cluster at CU Boulder, there is a different recommended

way of requesting an interactive shell. The sysadmins have written their own script to do it, called `sinteractive`. So, you could do:

```
sinteractive
```

On the Hummingbird cluster, at UCSC, it takes a few more steps. The webpage at https://www.hb.ucsc.edu/getting-started/ tells you how to do that. Go to that page and search for `Job Allocation - Interactive, Serial`. That takes you to the instructions. Minimally, here is what it looks like to get one core on a node on the Instruction partition, for 1 hour, with 1 Gigabyte of memory allocated to you:

```
[~]--% salloc --partition=Instruction --nodes=1 --time=01:00:00 --mem=1G --cpus-per-task=1
salloc: Granted job allocation 48130
[~]--% ssh $SLURM_NODELIST

# check that you are on the right host.
[~]--% hostname
hbcomp-005.hbhpc.ucsc.edu
```

Regardless of the method used to request an interactive shell, when the command returns with a command prompt, you should always check the hostname to make sure that you are not, inadvertently, still on the login node. Do that with the `hostname` command:

```
hostname
```

It is also worth using your alias `myjobs` (see the last part of Section 6.3 to learn how to make such an alias) to check that your interactive shell session is listed amongst your running jobs.

Once you have obtained an interactive shell on a compute node by one of the above methods, it should behave like any other shell you work on. You can give commands with it and define shell variables, etc. In short, it is perfect for running through your scripts, line by line, to make sure that they are working.

When you are done using your interactive shell, you should logout from it. This can be done by typing `logout`, or by hitting the `d` key while holding down the `control` button (i.e. `<cntrl>-d`). This returns the core that you were using to the pool of cores that can be allocated to other users.

Different HPCCs have different settings for the default amount of time an interactive shell will be granted to a user. On some systems, you can request a shell for a certain amount of time using the `--time` option to the slurm command.

6.4.2 Batch jobs

After you have tested your bioinformatic scripts in an interactive shell and are ready to run them "for real," you will want to launch the script in a non-interactive, or *batch* job. Such a job is one that you submit to the job scheduler, and then, when resources for the job become available, the scheduler will launch the job without any futher interaction from you. The job will run until it completes (or fails, or runs out of time, etc.), all without you having to interact directly with the computing cluster after submitting the job.

As you might expect, the SLURM command used to submit batch jobs is named (…wait for it!…) `sbatch`.
The syntax for using `sbatch` is:

```
sbatch sbatch_options script.sh script_arguments
```

where:

- `sbatch` is the command
- `sbatch_options` shows the place where different options to the `sbatch` command should be placed. These options can be a number of things like `--time=00:10:00` and `--mem=4G`. We will talk about `sbatch` options in more detail below.
- `script.sh` is the script that you have written that you want to run on the HPCC. We refer to this as the *job script*. It does not have to be named `script.sh`, but, since it is a shell script, it ought to have the `.sh` extension. The contents of this script file effectively define the *job* that you wish to run. It should be a shell script, and, on most SLURM systems you need to be specific about what "flavor" of shell it should be run in. This is done using the "shebang" line at the top of the script, like `#!/bin/bash` (See section ??). If you want to know what path should come after the `#!` (the "shebang"), you can give the command `which bash` from the shell on one of your cluster's login nodes (or an interactive shell on a compute node), and the shell should respond with the absolute path to the bash shell interpreter.
- `script_arguments` are any other options, or other arguments (like filenames) that you may want to pass to the script.

For practice in submitting batch jobs, we will, first write a simple job script that prints a few pieces of information about the shell that is running and the environment variables defined within that shell.

So, first, change into your scratch directory and make a new directory called `sbatch-play` and `cd` into it. Then, using a text editor, copy the following lines to a shell script called `easy.sh` and save it within your current working directory (`sbatch-play`). Note that the first line might have to be modified for your system, depending on the output of `which bash`, as mentioned above:

```
#!/bin/bash

echo "Starting at $(date)"
echo
echo "Current working directory is: $PWD"
echo "Current user is: $USER"
echo "Current hostname is: $(hostname)"
echo
echo "Currently, the PATH is set to: $PATH"
echo

# now, let's print the states of all the the environment variables
# and send them to stderr:
env > /dev/stderr

sleep 180
echo "Done at $(date)"
```

Once you have made that file, schedule it to run, telling the scheduler to put it in the queue to run, *and* that you are asking for only 5 minutes of computer time to run it, and to send *stdout* from the job to a file named `output-XXXXX`, and *stderr* to a file named `error-XXXXX`, where `XXXXX` is the job number that was assigned to your job. Job numbers get assigned in series by the job scheduler, so if the number is 4373238, then yours is the 4,373,238-th job that has been requested on the cluster.

```
sbatch --time=00:05:00 --output=output-%j --error=error-%j  easy.sh
```

You can check with your `myjobs` alias to see if the job has been scheduled and/or if it is running. Once it is running, you should see two new files in your current working directory `output-XXXXX` and `error-XXXXX`.

First, check the `output-XXXXX` file, using the `cat` command. You should see that the current working directory printed by the `easy.sh` script is the same as the current working directory from which you scheduled the job. Additionally, the PATH variable accessed inside the `easy.sh` script is the same as the PATH variable in the shell that you scheduled the job from. You can check this by issuing:

```
echo $PATH
```

in the shell from which you scheduled the job.

The take-home message from this little exploration is that the shell in which your job script, `easy.sh`, gets executed, inherits some of the important en-

vironment variables (PATH, PWD, etc.) that are in effect in the shell from which you scheduled the job.

However, a number of new environment variables have also been defined, most of them supplying extra information from SLURM itself. To see that, we can compare the file `error-XXXXX` (which holds information about all the environment variables in effect when the job scheduler launched `easy.sh`) to the output of the `env` command (which prints the values of all the environment variables) when we invoke it in the shell from which the job was scheduled. Try doing that like this:

```
# first, make a file of the environment variables in effect in the
# shell the job was scheduled from.  Sort these alphabetically to
# make it easier to compare between the two environments
env | sort > sheduling_env.txt

# also, sort the error-XXXXX file
sort error-XXXXX > slurm_running_env.txt

# then use less to compare sched_env.txt and error-XXXXX.

# The adventurous can use tmux to split their screen vertically into
# two panes to make this easier:
```

Sorting the files, as we have done above, garbles up some multi-line bash functions, but still makes it easy to see how the files differ. In particular, the large block of environment variables that start with `SLURM_` give information about the job resources and job numbers, etc. These can come in handy, though we will end up being more concerned with some of those variables when we start using SLURM *job arrays*.

6.4.2.1 The `sbatch --test-only` option

Especially when you start using an HPCC, scheduling jobs can be a confusing and somewhat daunting process. I used to always find my blood pressure going up when I submitted large jobs. Questions swirled in my head: "Will my script run? How long will I have to wait for my job to launch? Have I requested more resources than are available on the cluster such that my job will never launch?", etc. SLURM provides the `--test-only` option to `sbatch` that can be helpful in this regard. When `sbatch` is run with the `--test-only` option, the scheduler verifies that your script looks like a shell script, and then gives you information about how many cores in how many nodes and in which partition the job will be scheduled for. Try it like this:

```
sbatch --test-only --time=00:05:00 --output=output-%j --error=error-%j  easy.sh

# then see what happens if you request more the 24 hours:
sbatch --test-only --time=24:05:00 --output=output-%j --error=error-%j  easy.sh
```

As the name of the option implies, when you are using `--test-only`, your job *does not* actually get scheduled. `sbatch` with the `--test-only` option also gives an estimate of the time when your job would be launched. This estimate seems to be completely uninformative. It might tell you that it estimates your job would launch in 3 hours, but it could be much faster than that. One absolutely critical piece of information the `--test-only` option will give you is a nice big warning if you are requesting more resources than are available to you. For example, on the `shas` partition on SUMMIT, job run lengths are capped at 24 hours, so you can't request more than that amount of time. You can always use the `--test-only` option to ensure that you are not requesting more resources than are available.

We will note here that the `--test-only` option is somewhat akin to `rclone`'s `--dry-run` option, which lets you see what files would be copied, without actually copying them. This sort of option is always a nice feature when you want to check that you have set your requests up properly before committing computing or network resources to them.

6.4.2.2 Get messages from SLURM

If you want to be notified by email when a job starts and finishes, (or fails or gets killed by the scheduler). You can use the `--mail-user` option to supply an email address and the `--mail-type` option to tell the scheduler when it should email you with updates regarding your job status. For example, supply your own email address in the following, and try it:

```
sbatch --mail-type=ALL --mail-user=yourname@yourmail.edu --time=00:05:00 --output=output-%
```

You should receive an email from the cluster when your job starts and completes.

This can be a handy feature when you are not running too many jobs. If you have a large number of jobs, being notified every time a job starts and finishes can quickly fill up your inbox. In those cases, consider setting `--mail-type=FAIL` to only be notified when jobs have failed.

6.4.2.3 Include options *inside* your job script

As you might imagine, `sbatch` takes many different options. We list the main ones that you will be concerned with while doing bioinformatics in Table **??**. You might note that the above command line is getting quite long and cumbersome, and it would surely become very difficult to remember and type commands with even more options, every time you wanted to start a job. On top of that, the options that you would want to invoke are typically going to be specific to the job script that you are launching (i.e. `easy.sh` in the above example). Consequently, SLURM let's you specify the options passed to `sbatch` *inside the script file itself*. This leads to much simpler commands on the command line, and makes your workflows somewhat more reproducible. To include the `sbatch` options in the job script, you add them below the "shebang" line, in lines that start with `#SBATCH` before any other commands in the script.

For example, we could make a new file called `easy2.sh` like this:

```
#!/bin/bash

#SBATCH --time=00:05:00
#SBATCH --output=output-%j
#SBATCH --error=error-%j


echo "Starting at $(date)"
echo
echo "Current working directory is: $PWD"
echo "Current user is: $USER"
echo "Current hostname is: $(hostname)"
echo
echo "Currently, the PATH is set to: $PATH"
echo

# now, let's print the states of all the the environment variables
# and send them to stderr:
env > /dev/stderr

sleep 180
```

And we could then test it like this:

```
sbatch --test-only easy2.sh
```

or schedule it like this:

```
sbatch easy2.sh
```

6.4.2.4 The vagaries of `conda` within `sbatch`

As we noted above, the shell environment in which SLURM executes your scheduled job inherits many important variables (like PATH and the current working directory) from the shell that you submit the job from. One might hope that this would be sufficient to allow you to activate a conda environment from within your job script. This turns out not to be the case: even though several `conda` environment variables are set up and the path to `conda` is set, there are some `conda` initialization features that have to happen within your job script in order to be able to explicitly set your `conda` environment within your job script.

To see this, first make a script called `conda-test1.sh` that tries to activate different `conda` environments within the job script:

```
#!/bin/bash

#SBATCH --time=00:03:00
#SBATCH --output=conda-test1-output-%j
#SBATCH --error=conda-test1-error-%j


echo "Starting at $(date)"
echo
echo "Current working directory is: $PWD"
echo "Current user is: $USER"
echo "Current hostname is: $(hostname)"
echo
echo "Currently, the PATH is set to: $PATH"
echo
echo "Activating environment bioinf"
conda activate bioinf
echo
echo "Now the PATH is: $PATH"

sleep 120
```

and then run that with:

```
sbatch conda-test1.sh
```

After that has run, reading the contents of the `conda-test1-error-XXXX` file you should see an error about not being able to activate a conda environment. Likewise, if you look at the two `PATH`s printed out in `conda-test1-output-XXXX`, you will see that they are the same. So, indeed, the `bioinf` environemnt was not activated.

To remedy this problem, you need to source your `~/.bashrc` or your `~/.bash_profile` (whichever one contains the "conda init" block). If you set your `~/.bash_profile` up, as recommended in the beginning of the book (Section XXX) to read from your `~/.bashrc`, then you can simply source your `~/.bash_profile` in your job script.

For example, create a new file called `conda-init2.sh` and fill it with the following:

```
#!/bin/bash

#SBATCH --time=00:03:00
#SBATCH --output=conda-test2-output-%j
#SBATCH --error=conda-test2-error-%j

source ~/.bash_profile

echo "Starting at $(date)"
echo
echo "Current working directory is: $PWD"
echo "Current user is: $USER"
echo "Current hostname is: $(hostname)"
echo
echo "Currently, the PATH is set to: $PATH"
echo
echo "Activating environment bioinf"
conda activate bioinf
echo
echo "Now the PATH is: $PATH"
echo
echo "And the environment variables look like:"
echo "================================="
env

sleep 120
```

Then run it with:

```
sbatch conda-test2.sh
```

If `conda` is working properly within the job script there should be no output to `conda-test2-error-XXXX` and you should see that invoking `conda activate bioinf` changed the PATH variable to include paths from the `bioinf` environment.

The above points are really key if you want to use software like `bwa`, `samtools`, and `bcftools`, that you installed using Miniconda, from within a job script.

To repeat: to activate a conda environment like `bioinf` from within a job script running under `sbatch` you must add these lines:

```
source ~/.bash_profile
conda activate bioinf
```

Or, if your `~/.bash_profile` and `~/.bashrc` are set up differently, *and* your `conda` initialization block is in `~/.bashrc`, you should use:

```
source ~/.bashrc
conda activate bioinf
```

6.4.2.5 Test a longer running bash script

If you have time and want to pull this all together. Navigate to the `chinook-play` directory that we experimented in before. There should be a directory called `fastq` in there, as well as `chinook-genome-idx`.

Then, make a job script called `map-it.sh` with these lines in it (modifying the email to be your email address):

```
#!/bin/bash

#SBATCH --time=03:00:00
#SBATCH --output=map-it-output-%j
#SBATCH --error=map-it-error-%j
#SBATCH --mail-type=ALL
#SBATCH --mail-user=yourname@yourmail.edu


source ~/.bash_profile  # or ~/.bashrc if that is how you are set up
conda activate bioinf
```

```
# map it, then make it a BAM then run fixmate
bwa mem chinook-genome-idx/GCA_002872995.1_Otsh_v1.0_genomic.fna.gz \
  fastq/Battle_Creek_01_chinook_R1.fq.gz  \
  fastq/Battle_Creek_01_chinook_R2.fq.gz  | \
  samtools view -b -1 - | \
    samtools fixmate -m -O BAM - OUTPUT-fixed.bam

# now coordinate-sort it and mark PCR duplicates
samtools sort OUTPUT-fixed.bam | \
  samtools markdup - OUTPUT-marked.bam
```

And schedule it to run (from within the `chinook-play` directory) with:

```
sbatch map-it.sh
```

6.5 PREPATION INTERLUDE: An in-class exercise to make sure everything is configured correctly

Here, we run through a number of steps to start aligning sequence data to a reference genome. The main purpose of this exercise is to ensure that everyone has their systems configured correctly. Namely, we want to make sure that:

- You can log in to your cluster (SUMMIT, Hummingbird, etc.)
- You can use a few SLURM (job scheduler) commands.
- You can find your way to your *scratch* space on SUMMIT (or Hummingbird). *scratch* is fast, largely unlimited short-term storage. It is where you will do the majority of your bioinformatics.
- You can use `rclone` to tranfer files from Google Drive to your cluster. Doing things this way will allow you to access your Lab's Team Drive. The model for bioinformatic work is:
    a. Copy data from your Lab's Google Team Drive to *scratch* on the cluster.
    b. Do analyses of the data (in *scratch*)
    c. Copy the results back to your Lab's Google Team Drive before it gets deleted (by the system administrators) off of *scratch* Note: today you are not using your lab's Team drive, you are using the "class" shared drive, and you won't be copying anything back to it. But this is a start at least…

- You have Miniconda installed, that you have a conda environment with bioinformatic tools in it, and that you can activate that environment and use the tools within it. Miniconda makes it easy to install software that you need to run analyses.

Being able to successfully do all these things will be necessary to complete the next homework assignment which involves aligning short reads to a reference genome and will be assigned after spring break.

1. If you are not already in a tmux session, start a new `tmux` session called `inclass` on the login node.

```
tmux new -s inclass
```

2. Get a bash shell on a compute node

```
# on summit:
sinteractive

# on hummingbird
salloc --partition=Instruction --nodes=1 --time=02:00:00 --mem=4G --cpus-per-task=1
ssh $SLURM_NODELIST

# otherwise, you can try this (will work on Sedna)
srun --pty /bin/bash
```

3. Check what host you are on with `hostname`

```
hostname

# The hostname should be something like shasXXX.
#  Do not proceed if you are on a login node still (ie. hostname is loginXX)
```

   If you are still on a login node, try running `sinteractive` again.

4. Use your alias `myjobs` (if you have created it) to see what jobs you have running.

```
myjobs
```

```
# or, if you don't have that aliased:
squeue -u your_username

# You should see that you have one job running.  That "job" is a bash shell
# that you are interactively working with.
```

5. Navigate to your scratch directory using `cd`. On SUMMIT that will look like this:

```
cd /scratch/summit/wcfunk\@colostate.edu/
```

You can make a symbolic link to that in your home directory like this:

```
cd  # returns you to hour home directory

# this line makes a symbolic link called scratch in your home directory
# that points to your scratch space at /scratch/summit/wcfunk\@colostate.edu/
ln -s /scratch/summit/wcfunk\@colostate.edu/ scratch
```

Just be sure to use your own user name instead of wcfunk. Once that symbolic link is set up, you can `cd` into it, just like it was your scratch directory:

```
cd scratch
```

But if you don't want to set up a symbolic link just use the full path:

```
cd /scratch/summit/wcfunk\@colostate.edu/
```

On hummingbird you go to

```
cd /hb/scratch/username/
```

where username is your ucsc username. Note that if you want to

make a symbolic link to that, then *while in your home directory* do this:

```
ln -s /hb/scratch/username scratch
```

6. Within your account's scratch space, make a directory called `chinook-play` and `cd` into it:

```
mkdir chinook-play
cd chinook-play/
```

7. Make sure you have followed the link in your email to the shared google drive folder I sent you last night. If the email went to an email address that is not associated with your rclone configuration for google drive, then request access (at google drive) for that email. (Or setup a new rclone config for your other gmail address…)
8. Now, use rclone to list the files in the folder I shared with you:

```
rclone lsd --drive-shared-with-me gdrive-pers:CSU-con-gen-2020
```

   Note that `gdrive-pers` above is the name of my rclone configuration associated with the gmail account where I got an email (from myself at my CSU account) about the shared folder. You will have to change `gdrive-pers` to be appropriate to your config, depending on how you set it up. Note also how you access folders shared with you using `--drive-shared-with-me`.
9. After you get the above command to work, you should see something like the following, though it will change as more things get added to this directory…

```
-1 2020-03-09 17:51:06        -1 chr32-160-chinook
-1 2020-03-05 07:33:21        -1 pre-indexed-chinook-genome
```

10. Use rclone to download `big-fastq-play.zip` (about 170 MB). Once again you have to change the name of the rclone remoate (`gdrive-pers` in this case), to the name of your Google drive rclone remote.

```
rclone copy -P  --drive-shared-with-me gdrive-pers:CSU-con-gen-2020/big-fastq-play.z
```

11. Unzip the file you just downloaded. Note: on Hummingbird, `unzip` is not installed by default. So I recommend doing `conda install unzip` to add it to your base Miniconda environment.

```
unzip big-fastq-play.zip
```

    This creates a directory called `big-fastq-play` within your `chinook-play` directory.

12. Create a directory called `fastq` and move the two paired-end FASTQ files from `big-fastq-play` into it:

```
mkdir fastq
mv big-fastq-play/data/Battle_Creek_01_chinook_R* fastq/
# then make sure that they got moved
ls fastq
# if the files were moved, then feel free to remove the big fastq stuff
rm -r big-fastq-play*
```

13. Download the Chinook salmon genome into a directory called `genome`

```
mkdir genome
cd genome
wget ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/vertebrate_other/Oncorhynchus_tshaw
```

    That dude is 760 Mb, but it goes really quickly (at least on SUMMIT. Hummingbird seems to be a different story…)

14. Activate your `bioinf` conda environment to be able to use `bwa`:

```
conda activate bioinf
```

15. Index the genome with `bwa` in order to align reads to it:

```
bwa index GCA_002872995.1_Otsh_v1.0_genomic.fna.gz
```

This will take about 10 to 15 minutes. During that time, do this:

- If you are doing this within a `tmux` session, create a new tmux window (`cntrl-b c`). Note that this shell opens on the login node (the one that you started your tmux session on.)
- Rename this new tmux window "browse" (`cntrl-b ,`)
- Navigate to `~/scratch/chinook-play/genome/` and list the files there. These new files are being created by bwa.
- Switch back to your original tmux window.

16. If `bwa` is still running, ask me some questions. Or maybe I will tell you more about how conda works…
17. If `bwa` is still running after that, ask me about *nested tmux sessions*.
18. Truth be told. Indexing this thing can take a long time (30 to 40 minutes on a single core, perhaps). So, instead, here is what we will do:
    a. Kill the current `bwa` job but issuing `cntrl-c` in the shell where it is running.
    b. Use `rclone` to get the genome and a `bwa` index for it from my Google Drive. We want to put the result in the `chinook-play` directory inside a directory called `chinook-genome-idx`. NOTE: Again, you will likely have to use your name for your rclone remote, rather than `gdrive-pers`.

```
cd ../ # move up to the chinook-play directory
rclone copy  -P  --drive-shared-with-me gdrive-pers:CSU-con-gen-2020/pre-indexed-chi
```

That thing is 4.6 Gb, but can take less than a minute to transfer. (Let's see if things slow down with everyone running this command at the same time.)

19. Now when you do `ls` you should see something like this:

```
(bioinf) [chinook-play]--% ls
chinook-genome-idx  fastq  genome
```

20. For a final hurrah, we will start aligning those fastqs to the chinook genome. We will put the result in a directory we create called `sam`.

To make it clearer what we are doing we will define some shell variables.:

```
mkdir sam
GENOME=chinook-genome-idx/GCA_002872995.1_Otsh_v1.0_genomic.fna.gz
FQB=fastq/Battle_Creek_01_chinook_R
SAMOUT=$(basename $FQB).sam

# for fun, try echoing each of those variables and make sure they look right.
# for example:
echo $FQB

# ...

# then, launch bwa mem to map those reads to the genome.
# the syntax is:
# Usage: bwa mem [options] <idxbase> <in1.fq> [in2.fq]
bwa mem $GENOME ${FQB}1.fq.gz ${FQB}2.fq.gz > sam/$SAMOUT 2>sam/$SAMOUT.error
```

21. Once that has started, it is hard to know anything is happening, because we have redirected both stdout and stderr to files. So, do what you did before, use tmux to go to another shell and then use `tail` or `less` to look at the output files, which are in `sam/Battle_Creek_01_chinook_R.sam` and `sam/Battle_Creek_01_chinook_R.sam.error` within the `chinook-play` directory in `scratch`. You might even count how many lines of alignments file have been formed:

```
awk '!/^@/' Battle_Creek_01_chinook_R.sam | wc
```

22. Note that, in practice, we would not usually save the `.sam` file to disk. Rather, we would convert it into a compressed `.bam` file right as it comes off of `bwa mem`, and maybe even sort it as it is coming off…

23. Note that this process might not finish before the end of class. That is not a huge problem if you are running the job within `tmux`. You can logout and it will still keep running.

However, as said before, most of the time you will run *batch* jobs rather than interactive jobs. I just wanted to first give everyone a chance to step through these tasks in an interactive shell on a compute node because that is crucial

when developing your scripts, and we wanted to make sure that everyone has rclone, and miniconda installed and working.

6.6 More Boneyard...

Here is some nice stuff for summarizing all the information from the different runs from the chinook-wgs project:

```
qacct -o eriq -b 09271925 -j ml | tidy-qacct
```

Explain scratch space and how clusters are configured with respect to storage, etc.

Strategies—break names up with consistent characters:

- dashes within population names
- underscores for different groups of chromosomes
- periods for catenating pairs of pops

etc. Basically, it just makes it much easier to split things up when the time comes.

6.7 The Queue (SLURM/SGE/UGE)

6.8 Modules package

6.9 Compiling programs without admin privileges

Inevitably you will want to use a piece of software that is not available as a module or is not otherwise installed on they system.

Typically these software programs have a frightful web of dependencies.

Unix/Linux distros typically maintain all these dependencies as libraries or

packages that can be installed using a `rpm` or `yum`. However, the simple "plug-and-play" approach to using these programs requires have administrator privileges so that the software can be installed in one of the (typically protected) paths in the root (like `/usr/bin`).

But, you can use these programs to install packages into your home directory. Once you have done that, you need to let your system know where to look for these packages when it needs them (i.e., when running a program or *linking* to it whilst compiling up a program that uses it as a dependency.

Hoffman2 runs CentOS. Turns out that CentOS uses `yum` as a package manager.

Let's see if we can install llvm using yum.

```
yum search all llvm # <- this got me to devtoolset-7-all.x86_64 : Package shipping all ava

# a little web searching made it look like llvm-toolset-7-5.0.1-4.el7.x86_64.rpm or devtoo
# might be what we want.  The first is a dependency of the second...
mkdir ~/centos
```

Was using instructions at https://stackoverflow.com/questions/36651091/how-to-install-packages-in-linux-centos-without-root-user-with-automatic-depen

Couldn't get yum downloader to download any packages. The whole thing looked like it was going to be a mess, so I thought I would try with miniconda.

I installed miniconda (python 2.7 version) into `/u/nobackup/kruegg/eriq/programs/miniconda/` and then did this:

```
# probably could have listed them all at once, but wanted to watch them go
# one at a time...
conda install numpy
conda install scipy
conda install pandas
conda install numba

# those all ran great.

conda install pysnptools

# that one didn't find a match, but I found on the web that I should try:
conda install -c bioconda pysnptools

# that worked!
```

Also we want to touch briefly on LD_PATH (linking failures—and note that libraries are often named libxxx.a) and CPATH (for failure to find xxxx.h), etc.

6.10 Job arrays

Quick note: Redefine IFS to break on TABs so you can have full commands in there. This is super useful for parsing job-array COMMLINES files.

```
IFS=$'\t\n'; BOP=($(echo boing | awk '{printf("first\tsecond\tthird that is long\tfourth\n
```

Definitely mention the `eval` keyword in bash for when you want to print command lines with redirects.

Show the routine for it, and develop a good approach to efficiently orchestrating redos. If you know the taskIDs of the ones that failed then it is pretty easy to write an awk script that picks out the commands and puts them in a new file. Actually, it is probably better to just cycle over the numbers and use the -t option to launch each. Then there is now changing the job-ids file.

In fact, I am starting to think that the -t option is better than putting it into the file.

Question: if you give something on the command line, does that override the directive in the header of the file? If so, then you don't even need to change the file. Note that using the qsub command line options instead of the directives really opens up a lot of possibilities for writing useful scripts that are flexible.

Also use short names for the jobs and have a system for naming the redos (append numbers so you know which round it is, too) possibly base the name on the ways things failed the first time. Like, `fsttf1` = "Fst run for things that failed due to time limits, 1". Or structure things so that redos can just be done by invoking it with -t and the jobid.

6.11 Writing stdout and stderr to files

This is always good to do. Note that `stdbuf` is super useful here so that things don't get buffered super long. (PCAngsd doesn't seem to write antyhing till the end…)

6.12 Breaking stuff down

It is probably worth talking about how problems can be broken down into smaller ones. Maybe give an example, and then say that we will be talking about this for every step of the way in bioinformatic pipelines.

One thing to note—sometimes processes go awry for one reason or another. When things are in smaller chunks it is not such a huge investment to re-run it. (Unlike stuff that runs for two weeks before you realize that it ain't working right).

Part II

Part II: Reproducible Research Strategies

7

Introduction to Reproducible Research

(Pritchard et al., 2000)

And let's again try to pitch it in there: (Pritchard et al., 2000).

Let's try chucking it in without parens: Pritchard et al. (2000)

8

Rstudio and Project-centered Organization

Somewhere talk about here::here(): https://github.com/jennybc/here_here

8.1 Organizing big projects

By "big" I mean something like the chinook project, or your typical thing this is a chapter in a dissertation or a paper.

I think it is useful for number things in order on a three-digit system, and at the top of each make directories `outputs` and `intermediates`, like this:

```
dir.create(file.path(c("outputs", "intermediates"), "203"), recursive = TRUE, showWarnings
```

I had previously used two variables `output_dir` and `interm_dir` to specify these in each notebook, but now I think it would be better to just hardwire those, for a few reasons:

- Sometimes you are working on two notebooks at once in the same environment and you don't want to get confused about where things should get written.
- You can't use those variables in shell blocks of code, where you will just have to write the paths out anyway.
- Hard-wiring the paths forces you to think about the fact that once you establish the name for something, you should not change it, ever.
- Hard-wiring the paths makes it easy to identify access to those different files. In particular you can write an R script that rips through all the lines of code in the Rmds (and R files) in your project and records all the instances of writing and reading of files from the outputs and intermediates directories. If you do this, you can make a pretty cool dependency graph so that you can visualize what you need to keep to clean things up for a final reproducible project. *Note: I should write a little R package that can analyze such dependencies in a project. Unless there is already something like that. (Note*

that these are not package dependencies, but, rather, internal project dependencies. Note that if one is consistent with using readr functions it would be pretty easy to find all those instances of `read_*` *and* `write_*` *and that makes it clear why standardized syntax like that is so useful.* Hey! Notice that this type of analysis would be made simple if we just focused on dependencies between different Rmds. That is probably the level we want to keep it at as well. Ideally you can make a graph of all files that are output from one Rmd and read into another. That would be a fun graph to make of the Chinook project.

- Note. You should keep 900-999 as 100 slots for Rnotebooks for the final reproducible project to go with a publication. So, you can pare down all the previous notebooks and things.
- Hey! Sometimes you are going to want to write or read files that have been auto-produced. For example, if you are cycling over chromosomes, you might have output files that start something like: `outputs/302/chromo_output_`. So, when generating those names, make sure that the full prefix is in there, and has a trailing underscore. Then you can still find it with a regex search, and also recognize it as a signifying a class of output files.

9

Version control

10

A fast, furious overview of the tidyverse

Basically want to highlight why it can be so useful for bioinformatic data (and also some of the limitations with really large data sets).

(But, once you have whittled bams and vcfs down to things like GWAS results and tables of theta values, they dplyr is totally up for the job.)

A really key concept here is going to be the relational data model (e.g. tidy data) and how it is so much better for handling data.

A superpowerful example of this is provided by tidytree[1] which allows one to convert from phylo objects to a tidy object: Shoot! that is so much easier to look at! This is a great example of how a single approach to manipulating data works so well for other things that have traditionally not been manipulated that way (and as a conseqence have been completely opaque for a long time to most people.)

Cool. I should definitely have a chapter on tidy trees and ggtree.

What I really want to stress is that the syntax of the tidyverse is such that it makes programming a relaxing and enjoying experience.

[1] https://cran.r-project.org/web/packages/tidytree/vignettes/tidytree.html

11

Authoring reproducibly with Rmarkdown

11.1 Notebooks

Here is a pro-tip. First, number your notebooks and have outputs and intermediates directories associated with them. And second, always save the R object that is a ggplot in the outputs so that if you want to tweak it without re-generating all the underlying data, you can do that easily.

11.2 References

Science, as an enterprise, has proceeded with each new generation of researchers building upon the discoveries and achievements of the previous. Scientific publication honors this tradition with the stringent requirement of diligent citation of previous work. Not only that, but it is incumbent upon every researcher to identify all current work by others that is related to their own, and discuss its similarities and differences. As recently as the early 90s, literature searches involved using an annual index *printed on paper!* And, if you found a relevant paper you had to locate it in a bound volume in the library stacks and copy it page by page at a Xerox machine (or send an undergraduate intern to do that…)

Today, of course, the Internet, search services like Google Scholar, and even Twitter, have made it far easier to identify related work and to keep abreast of the latest developments in your field. But, this profusion of new literature leads to new challenges with managing all this information in a way that makes it easy for you to access, read, and cite papers while writing your own publications. There are many reference management systems available today, to help you with this task. Some of these are proprietary and paid products like EndNote. Many institutions (like Colorado State University) have licenses that provide their students a no-cost way to obtain EndNote, but the license

will not extend to updates to the program, once the student has graduated. (CHECK THIS!!!).

An alternative citation manager is Zotero. It is an open source project that has been funded not by publishing companies (like its non-open-source competitors, Mendeley and ReadCube) but by the non-profit Corporation for Digital Scholarship. As an open-source project, outside contributors have been enabled to develop workflows for integrating Zotero with reproducible research authoring modalities like Rmarkdown, including RStudio integration that lets you drop citations from Zotero directly into your Rmarkdown document where they will be cited and included in the references list in the format of just about any journal you might want to choose from. Accordingly, I will describe how to use Zotero as a citation manager while writing Rmarkdown documents.

Install Zotero and be sure to install the connector for Chrome, Firefox, or Safari

11.2.1 Zotero and Rmarkdown

Zotero has to be customized slightly to integrate with Rmarkdown and Rstudio, you must install the R package `citr`, and you should make some configurations:

1. First, you have to get a Zotero add-on that can translate your Zotero library into a different format called BibTeX format (which is used with the TeX typesetting engine and the LaTeX document preparation system). Do this by following the directions at https://retorque.re/zotero-better-bibtex/installation/.
2. When you restart Zotero, you can choose all the default configurations in the BetterBibTeX start-up wizard.
3. Then, configure the BetterBibTex preferences by going to the Zotero preferences, choosing BetterBibTex, and then selecting "Export" button. That yields a page that gives you a place to omit certain reference fields. You life will be easier if you omit fields that are often long, or which are not needed for citation. I recommend filling that field with:

   ```
   abstract,copyright,file,pmid
   ```

   And, you probably should restart Zotero after doing that.
4. Install the R package `citr`. It is on CRAN, but it is probably best

to first try the latest development version. Install it from within R using:

```
devtools::install_github("crsh/citr")
```

For more information about this package check out https://github.com/crsh/citr.

5. Once that package is installed. Quit and re-open RStudio. Now, if you go to the "Addins" menu (right under the name panel at the top of the RStudio window) you will see the option to "Insert citations." Choosing that brings up a dialog box. You can choose the Zotero libraries to connect to. It might take a while to load your Zotero library if it is large. Once it is loaded though, you just start typing the name of the author or part of an article title, and *boom!* if that article is in your library it appears as an option. If you select it, you get a markdown citation in your text.
6. To avoid having to go to the "Addins" menu, you can set a keyboard shortcut for "Insert citations" by choosing the "Code" section of RStudio's preferences and, under the "Editing" tab, clicking the "Modify Keyboard Shortcuts" command, searching for "Insert citations" and then selecting the keyboard shortcut area of the row and keying in which keys you would like to give you the shortcut (for example, Shift-CMD-I).

After those steps, you are set up to draw from your Zotero library or libraries to insert citations into your R markdown document.

Pretty cool, but there are some things that are sort of painful—namely the Title vs. Sentence casing. Fortunately, citr just adds things to your references.bib, it doesn't re-overwrote references.bib each time, so you can edit references.bib to put titles in sentence case. Probably want to export without braces protecting capitals. Then it should all work. See this discussion[1]. Just be sure to version control references.bib and commit it often. Though, you might want to go back and edit stuff in your Zotero library.

(Barson et al., 2015)

[1] https://forums.zotero.org/discussion/61715/prevent-extra-braces-in-bibtex-export

11.3 Bookdown

Whoa! Bookdown has figured out how to do references to sections and tables and things in a reasonable way that just isn't there for the vanilla Rmarkdown. But you can use the bookdown syntax for a non-book too. Just put something like this in the YAML:

```
output:
  bookdown::word_document2: default
  bookdown::pdf_document2:
    number_sections: yes
  bookdown::html_document2:
    df_print: paged
```

11.4 Google Docs

This ain't reproducible research, but I really like the integration with Zotero. Perhaps I need a chapter which is separate from this chapter that is about disseminating results and submitting stuff, etc.

12

Using python

Many people live and breathe python. Most conservation geneticists and most biologists in general are probably more familiar and comfortable with R. Nonetheless, there are some pieces of software that are available in python and it behooves us to know at least enough to crack those open and start using them.

Happily, the makers of RStudio have made it very easy to use python within the RStudio IDE. This makes the transition considerably easier.

Topics:

- conda and installing and setting up python environments. (What are they under the hood?)
- the reticulate package
- step-by-step instructions.
- An example: 2Dsfs and the moments package.

Part III

Part III: Bioinformatic Analyses

13

Overview of Bioinformatic Analyses

This is going to be sequence based stuff and getting to variants.

Part IV will be "bioinformaticky" analyses that you might encounter once you have your variants.

I think now that I will insert QC relevant to each step in that section.

Hey Eric! Find a place to put a section about bedtools in there. It is pretty cool what you can do with it, even on VCF files, like coverage of a VCF file in windows to get a sliding window of variant density.

14

DNA Sequences and Sequencing

To understand the fundamentals of alignment of DNA sequence to a reference genome, and all the intricacies that such a process entails, it is important to know a few important facts about DNA and how the sequence of a DNA molecule can be determined. This section may be review for some, but it presents the minimal set of knowledge needed to understand how DNA sequencing works, today.

14.1 DNA Stuff

In bioinformatics, we are primarily going to be interested in representing DNA molecules in text format as a series of nucleotide bases (A, C, G, T). For the most part, we don't want to stray from that very simple representation; however, it is important to understand a handful of things about *DNA directionality* and the action of DNA polymerase during the process of *DNA replication* to understand next generation sequencing and DNA alignment conventions.

DNA typically occurs as a double-helix of two *complementary* strands. Each strand is composed of a *backbone* of phophates and deoxyribose molecules, to which DNA *bases* are attached. Figure 14.1 shows this for a fragment of *double-stranded* DNA of four nucleotides.

The orange parts show the ribose molecules in the backbone, while the four remaining colors denote the nucleotide bases of adenine (A), cytosine (C), guanine (G), and thymine (T). Each base on one strand is associated with its *complement* on the other strand. The hydrogen bonds between complements is what holds the two strands of DNA together. A and T are complements, and C and G are complements. (I remember this by noting that C and G and curvy and A and T are sharp and angular, so the pairs go together…).

Immediately this raises two challenges that must be resolved for making a simple, text-based representation of DNA:

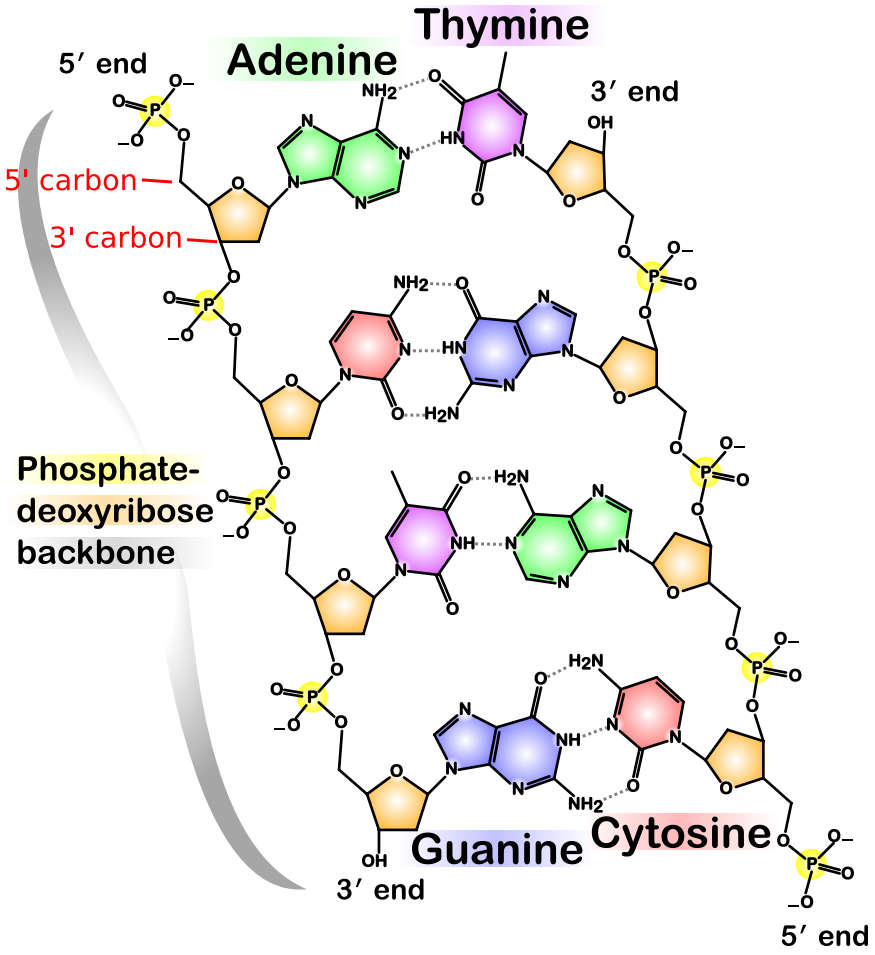


FIGURE 14.1: Schematic of the structure of DNA. (Figure By Madprime (talk—contribs) CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1848174)

1. When describing a sequence of DNA bases, which direction will we read it in?
2. Which strand will we read off of?

We leave the second question until later. But note that the order in which DNA sequence is read is, by convention, from the 5' to the 3' end of the molecule. The terms 3' and 5' refer to different carbon atoms on the ribose backbone molecules. In the figure above, each little "kink" in the ribose molecule ring (and in the attached lines) is a carbon molecule. If you count clockwise from the oxygen in ribose, you see that the third carbon (the 3' carbon) is the carbon atom that leads to a phosphate group, and through the phosphate, to an attachment with the 5' carbon atom of the next ribose. The 3' and 5' carbon atoms are labelled in red on the top-left ribose molecule in Figure 14.1.

Notice that when we speak of reading a DNA sequence, we are implicitly talking about reading the sequence of one of the two strands of the double-stranded DNA molecule. I'll say it again: a DNA sequence is always the sequence of one *strand* of the molecule. But, if that DNA were "out living in the wild" it would have been in double-stranded form, and would have been paired with its complement sequence. For now, just note that any DNA sequence always has a complement which is read in the reverse direction. Thus, if we have a sequence like:

```
5'--ACTCGACCT--3'
```

Then, paired with its complement it would look like:

```
5'--ACTCGACCT--3'
    |||||||||
3'--TGAGCTGGA--5'
```

and, if you were to write its complement in standard 5' to 3' order, you would have to reverse it like so:

```
5'--AGGTCGAGT--3'
```

14.1.1 DNA Replication with DNA Polymerase

So, why do we read DNA sequence from 5' to 3'? Is it just because geneticists are wacky, backwards folks and thought it would be fun to read in a direction that sounds, numerically, to be backwards? No! It is because $5' \longrightarrow 3'$ is the direction in which a new strand of DNA is synthesized during DNA replication.

When Watson and Crick (1953) published the first account of the double helical structure of DNA, they noted that the double-helix (i.e., two-stranded) nature of the molecule immediately suggested a copying mechanism (Figure 14.2).

It has not escaped our notice that the specific pairing we have postulated immediately suggests a possible copying mechanism for the genetic material.

Full details of the structure, including the conditions assumed in building it, together with a set of co-ordinates for the atoms, will be published elsewhere.

We are much indebted to Dr. Jerry Donohue for constant advice and criticism, especially on interatomic distances. We have also been stimulated by a knowledge of the general nature of the unpublished experimental results and ideas of Dr. M. H. F. Wilkins, Dr. R. E. Franklin and their co-workers at

FIGURE 14.2: Excerpt from Crick and Watson (1953).

(I've also included, in the excerpt, the inadequate acknowledgment of the centrality of Rosalind Franklin's pioneering X-ray crystallography results to Crick and Watson's conclusions—an issue in scientific history about which much has been written, see, for example, this article[1] in *The Guardian*.)

Figure 14.3 shows a schematic of what DNA looks like during the replication process.

Essentially, during DNA replication, a *DNA polymerase* molecule finds nucleotide bases (attached to three phoshpate groups) to build a new strand that is complementary to the DNA *template* strand, and it guides those *nucleotide triphosphates* to the appropriate place in the complementary strand and helps them be incorporated into that growing, complementary strand. The newly syntesized strand is a *reverse complement* of the template strand. However, DNA polymerase is not capable of "setting up shop" anywhere upon a template strand and simply stuffing complementary bases in wherever it wants. In fact, DNA polymerase is only able to add new bases to a growing strand if it can attach the new nucleotide triphosphate to a *free 3' hydroxyl* that is on the end of the growing strand (the 3' hydroxyl is just a hydroxyl group attached to the 3' carbon). In Figure 14.3, the template strand is the strand on the right of the figure, and the growing complementary strand is on the left side. There is a free 3' hydroxyl group on the ribose attached to the cytosine base. That is what is needed for DNA polymerase to be able to place a thymine triphosphate (complementary to adenine on the template strand) in the currently vacant position. If that thymine comes with a free 3' hydroxyl

[1] https://www.theguardian.com/science/2015/jun/23/sexism-in-science-did-watson-and-crick-really-steal-rosalind-franklins-data

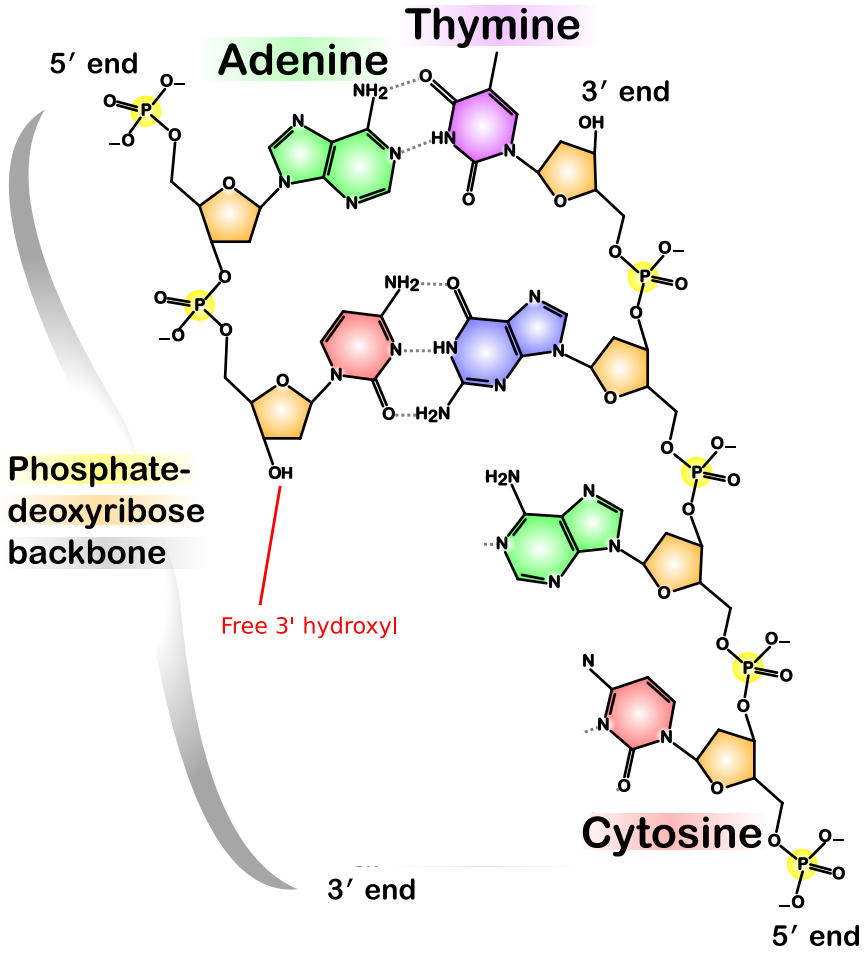


FIGURE 14.3: DNA during replication. (Figure adapted from the one by Madprime (talk—contribs) CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1848174)

group, then DNA polymerase will next place a guanine (complementary to the cytosine on the template strand) on the growing chain. And so forth. Thus, we see how the new strand of DNA is synthesized from the 5' to the 3' end *of the growing chain*.

Of course, some people find it easier to think about a new strand of DNA being sythesized in the 3' to 5' direction *along the template strand*. This is equivalent. However, if you just remember that "free three" rhymes, and that DNA polymerase needs a free 3' hydroxyl to add a new base to the growing strand, you can always deduce that DNA must "grow" in a 5' to 3' direction.

14.1.2 The importance of the 3' hydroxyl...

It would be hard to overstate the importance to molecular biology of DNA polymerase's dependence upon a free 3' hydroxyl group for new strand synthesis. This simple fact plays a central role in:

1. polymerase chain reaction (PCR)—the PCR primers are little oligonucleotides that attach to a template strand and provide a free 3' hydroxyl group for the initiation of synthesis.
2. a ddNTP is a nucleotide attached to a ribose molecule that lacks a hydroxyl group on its 3' carbon. Incorporation of such a ddNTP into a growing DNA strand terminates further DNA extension, and forms the basis for Sanger sequencing (we'll explore this below).
3. Some medications are designed to interfere with viral DNA replication. For example, AZT, or azino-thymine, is an anti-retroviral drug used to slow the progression of AIDS. It is a thymine nucleotide with an azino (N_3) group (instead of a hydroxyl group) attached to the 3' carbon. Azino-thymine is used preferentially by reverse transcriptase when synthesizing DNA. Incorporation of it into a growing chain terminates DNA synthesis.

The reversible inhibition of DNA extension also plays an important role in sequencing by synthesis as used by Illumina platforms. We will discuss this in a moment, but first we take a stroll down memory lane to refresh our understanding of *Sanger sequencing* so as to understand how radically different *next-generation sequencing* technologies are.

14.2 Sanger sequencing

It is hard to imagine that the first public human genome was sequenced almost entirely by Sanger sequencing. We discuss the Sanger sequencing method here so we can contrast it with what happens on, say, an Illumina machine today.

To perform Sanger sequencing, first it was necessary to do PCR to create numerous copies of a double stranded DNA fragment that was to be sequenced. For example let's say that one wanted to sequence the 20-mer shown below, represented as double stranded DNA.

```
5'--AGGCTCAAGCTTCGACCGT--3'
3'--TCCGAGTTCGAAGCTGGCA--5'
```

For Sanger sequencing, first, one would do PCR to create bazillions of copies of that double-stranded DNA. Then four separate further PCR reactions would be done, each one having been "spiked" with a little bit of one of four different ddNTPs which, if incorporated into the growing strand allow no further extension of it.

For example, if PCR were done as usual, but with the addition of ddATP, then occasionally, when a ddATP (an A lacking a 3' hydroxyl group) is incorporated into the growing strand that strand will grow no more. Consequently, the products of that PCR (incorporating an appropriate concentration of ddATPs), once filtered to retain only the top strand from above, will include the fragments

```
## [1] "A*"               "AGGCTCA*"
## [3] "AGGCTCAA*"        "AGGCTCAAGCTTCGA*"
```

Where the * follows the sequence-terminating base.

Likewise, in a separate reaction, occasional incorporation of a ddCTP will yield products:

```
## [1] "AGGC*"              "AGGCTC*"
## [3] "AGGCTCAAGC*"        "AGGCTCAAGCTTC*"
## [5] "AGGCTCAAGCTTCGAC*"  "AGGCTCAAGCTTCGACC*"
```

And in another reaction, occasional incorporation of ddGTP yields:

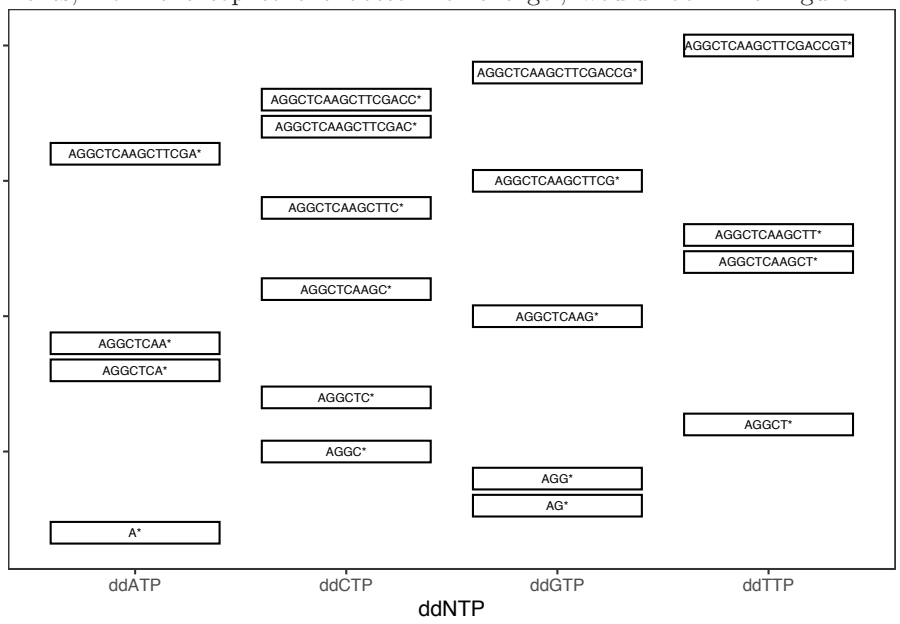
```
## [1] "AG*"                 "AGG*"
## [3] "AGGCTCAAG*"          "AGGCTCAAGCTTCG*"
## [5] "AGGCTCAAGCTTCGACCG*"
```

And in a final, separate reaction, incorporation of ddTTP would give:

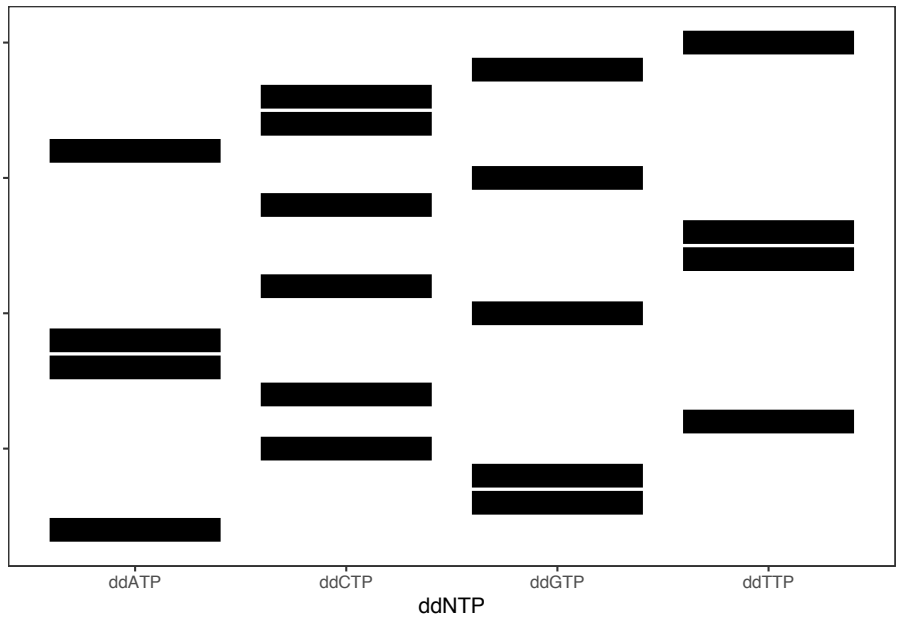
```
## [1] "AGGCT*"               "AGGCTCAAGCT*"
```

```
## [3] "AGGCTCAAGCTT*"          "AGGCTCAAGCTTCGACCGT*"
```

Each of these fragments is of a different length, so they can be separated on an electrophoretic gel. The secret to Sanger sequencing is that the products of all four separate reactions can be run side by side in separate lanes of the gel (or, as was done later, in separate capillaries…), so that the gel, with small fragments running faster than large fragments, from the top to the bottom of the gel, would look like Figure **??**:



In reality, a gel like that in Figure **??**, won't bear the name of each sequence fragment. In fact, it will look much more like what you see in Figure **??**.



However, even with the DNA fragemnt sequences obscured, their sequences can be detemined by working from the bottom to the top and adding a different DNA base according to which column the band is in. Try it out.

Some very important points must be made about Sanger sequencing:

1. The signal obtained from sequencing is, in a sense, a mixture of the starting templates. So, if you have DNA from an individual, you have two copies of each chromosome, and they might carry slightly different sequences. At heterozygous sites, it is impossible to tell which allele came from which chromosome.
2. To conduct this procedure, it was typical that specific PCR primers were used to sequence only a single fragment of interest. Extending the sequencing beyond that region often involved a laborious process of "walking" primers further out in either direction. Very tedious.
3. Each sequencing reaction was typically carried out for just a single individual at once.

4. Until a little over a decade ago, this is how sequencing was done in conservation genetics.

14.3 Illumina Sequencing by Synthesis

Illumina paired-end sequencing is currently the leading technology in conservation genomics.

They say that a picture is worth a thousand words, so a video may well be worth ten thousand. Illumina has a very informative video about sequencing by synthesis.

I have used some code (I found on GitHub) to provide captions to the video. These captions include comments, as well as questions that form part of this week's homework. (Yay!)
You can see the video at https://eriqande.github.io/erics-captioned-vids/vids/illumina-sbs/

The main take-home messages I want everyone to get about Illumin sequencing is:

1. The signal obtained from each cluster is the sequence of a single, *single-stranded DNA fragment*
2. In paired-end sequencing, sequence from both ends of the fragment (but not necessarily the middle) is obtained.
3. The technique lends itself to sequencing millions of "anonymous" chunks of DNA.
4. The indexes, or "barcodes" allow DNA from multiple individuals to be sequenced in a single run.
5. This is how most high-throughput sequencing is done today.

14.4 Library Prep Protocols

Gotta mention here about how barcodes work.

How you prepare your libraries dictates what type of data you get.

14.4.1 WGS

14.4.2 RAD-Seq methods

14.4.3 Amplicon Sequencing

14.4.4 Capture arrays, RAPTURE, etc.

15

Bioinformatic file formats

Almost all the high-throughput sequencing data you will deal with should arrive in just a few different formats. There are some specialized formats (like those output by the program TASSEL, etc.) but we will largely ignore those, focusing instead on the formats used in production by the 1000 genomes and 10K vertebrate genomes projects. In a sentence, the most important are: FASTA, FASTQ, SAM, BAM, and VCF.

Plan: Go over these, and for each, pay special attention to compressed and indexed forms and explain why that is so important. I think that we should probably talk about the programs that are available for manipulating each of these, as well, but I won't add that until we all have access to an HPC or other proper Unix environment.

I was originally going to do tools for manipulating files in the different formats, but I will do that in a separate chapter(s) later.

15.1 Sequences

15.2 FASTQ

The FASTQ format is the standard format for lightly-processed data coming off an Illumina machine. If you are doing whole genome sequencing, it is typical for the Illumina processing pipeline to have separated all the reads with different barcodes into different, separate files. Typically this means that all the reads from one library prep for one sample will be in a single file. The barcodes and any associated additional adapter sequence is typically also removed from the reads. If you are processing RAD data, the reads may not be separated by barcode, because, due to the vagaries of the RAD library prep, the barcodes might appear on different ends of some reads than expected.

A typical file extension for FASTQ files is `.fq`. Almost all FASTQ files you get

from a sequencer should be gzipped to save space. Thus, in order to view the file you will have to uncompress it. Since you would, in most circumstances, want to visually inspect just a few lines, it is best to do that with `gzcat` and pipe the output to `head`.

As we have seen, paired-end sequencing produces two separate reads of a DNA fragment. Those two different reads are usually stored in two separate files named in such a manner as to transparently denote whether it contains sequences obtained from read 1 or read 2. For example `bird_08_B03_R1.fq.gz` and `bird_08_B03_R2.fq.gz`. Read 1 and Read 2 from a paired read must occupy the sames lines in their respective files, i.e., lines 10001-10004 in `bird_08_B03_R1.fq.gz` and lines 10001-10004 in `bird_08_B03_R2.fq.gz` should both pertain to the same DNA fragment that was sequenced. That is what is meant by "paired-end" sequencing: the sequences come in pairs from different ends of the same fragment.

The FASTQ format is *very* simple: information about each read occupies just four lines. This means that the number of lines in a proper FASTQ file must always be a multiple of four. Briefly, the four lines of information about each read are always in the same order as follows:

1. An Identifier line
2. The DNA sequence as A's, C's, G's and T's.
3. A line that is almost always simply a `+` sign, but can optionally be followed by a repeat of the ID line.
4. An ASCII-encoded, Phred-scaled base quality score. This gives an estimated measure of certainty about each base call in the sequence.

The code block below shows three reads worth (twelve lines) of information from a FASTQ file. Take a moment to identify the four different lines for each read.

```
@K00364:64:HTYYCBBXX:1:1108:4635:14133/1
TAGAATACGCCAGGTGTAAGAATAGTAGAATACGCCAGGTGTAAGAATAGTAGAACACGCCAGGTGTAAGAATAGTAGAA
+
AAAFFJJJJJFFJFJJJFJJFFJFJFJJJJJJJJFFJJJFJJJFJJAJJFJJFJJFJJJ7JJJF-<JAFJJ<F<AJAJJF
@K00364:64:HTYYCBBXX:1:1108:5081:25527/1
AAAACACCAAAAGAAAGATGCCCAGGGTCCCTGCTCATCTGCGTGAAAGTGACTTAGGCATGCTGCAAGGAGGCATGAGG
+
AAFFFJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
@K00364:64:HTYYCBBXX:1:1108:5852:47295/1
AGGTGGCTCTAGAAGGTTCTCGGACCGAGAAACAGCCTCGTACATTTGCAATGATTTCAATTCATTTTGACCATTACGAA
+
AAFFFJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
```

Lines 2 and 3 are self-explanatory, but we will expound upon lines 1 and 4 below.

15.2.1 Line 1: Illumina identifier lines

The identifier line can be just about any string that starts with an `@`, but, from Illumina data, today, you will typically see something like this:

```
@K00364:64:HTYYCBBXX:1:1108:4635:14133/1
```

The colons (and the `/`) are field separators. The separate parts of the line above are interpreted something along the lines as follows (keeping in mind that Illumina occasionally changes the format and that there may be additional optional fields):

```
@           : mandatory character that starts the ID line
K00364      : Unique sequencing machine ID
64          : Run number on instrument
HTYYCBBXX   : Unique flow cell identifier
1           : Lane number
1108        : Tile number (section of the lane)
4635        : x-coordinate of the cluster within the tile
14133       : y-coordinate of the cluster within the tile
1           : Whether the sequence is from read 1 or read 2
```

Question: For paired reads, do you expect the x- and y-coordinates for read 1 and read 2 to be the same?

15.2.2 Line 4: Base quality scores

The base quality scores give an estimate of the probability that the base call is incorrect. This comes from data the sequencer collects on the brightness and compactness of the cluster radiance, other factors, and Illumina's own models for base call accuracy. If we let p be the probability that a base is called incorrectly, then Q, the Phred-scaled base quality score, is:

$$Q = \lfloor -10 \log_{10} p \rfloor,$$

where $\lfloor x \rfloor$ means "the largest integer smaller than x.

To get the estimate of the probability that the base is called incorrectly from the Phred scaled score, you invert the above equation:

$$p = \frac{1}{10^{Q/10}}$$

Base quality scores from Illumina range from 0 to 40. The easiest values to use as guideposts are $Q = 0, 10, 20, 30, 40$, which correspond to error probabilites of 1, 1 in 10, 1 in 100, 1 in 1,000, and 1 in 10,000, respectively.

All this is fine and well, but when we look at the quality score line above

we see something like `7JJJF-<JAFJJ<F<AJAJJF`. What gives? Well, from a file storage and parsing perspective, it makes sense to only use a single character to store the quality score for every base. So, that is what has been done: each of those single characters represents a quality score—a number between 0 and 40, inclusive.

The values that have been used are the *decimal representations of ASCII text characters* minus 33.
The decimal representation or each character can be found in Figure 15.1.

The decimal representation is in the upper left of each character's rectangle.

Find the characters corresponding to base quality scores of 0, 10, 20, 30, and 40. Remember that the base quality score is the character's decimal representation *minus 33*.

Here is another question: why do you think the scale starts with ASCII character 33?

15.2.3 A FASTQ 'tidyverse' Interlude

Here we demonstrate some R code while exploring FASTQ files. First, in order to do these exercises you will want to download and launch the RStudio project, `big-fastq-play`, that has the data in it. Please work within that RStudio project to do these exercises. Here is a direct download link to it: https://docs.google.com/uc?export=download&id=1iD8tz_KSOHDBpXvXssqo3noPZAm1Qsjp

Note that this might stress out older laptops as it loads 100s of Mb of sequence data into memory.

15.2.3.1 Reading FASTQs with `read_lines`

Load packages:

```
library(tidyverse)

# if you don't have this package, get it
# install.packages("viridis")
library(viridis)
```

Read the FASTQ, make it a matrix, then make it a tibble

ASCII CONTROL CODE CHART

b7 b6 b5 / BITS b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0 0	16 DLE 10 20	32 SP 20 40	48 0 30 60	64 @ 40 100	80 P 50 120	96 ‘ 60 140	112 p 70 160
0 0 0 1	1 SOH 1 1	17 DC1 11 21	33 ! 21 41	49 1 31 61	65 A 41 101	81 Q 51 121	97 a 61 141	113 q 71 161
0 0 1 0	2 STX 2 2	18 DC2 12 22	34 " 22 42	50 2 32 62	66 B 42 102	82 R 52 122	98 b 62 142	114 r 72 162
0 0 1 1	3 ETX 3 3	19 DC3 13 23	35 # 23 43	51 3 33 63	67 C 43 103	83 S 53 123	99 c 63 143	115 s 73 163
0 1 0 0	4 EOT 4 4	20 DC4 14 24	36 $ 24 44	52 4 34 64	68 D 44 104	84 T 54 124	100 d 64 144	116 t 74 164
0 1 0 1	5 ENQ 5 5	21 NAK 15 25	37 % 25 45	53 5 35 65	69 E 45 105	85 U 55 125	101 e 65 145	117 u 75 165
0 1 1 0	6 ACK 6 6	22 SYN 16 26	38 & 26 46	54 6 36 66	70 F 46 106	86 V 56 126	102 f 66 146	118 v 76 166
0 1 1 1	7 BEL 7 7	23 ETB 17 27	39 ’ 27 47	55 7 37 67	71 G 47 107	87 W 57 127	103 g 67 147	119 w 77 167
1 0 0 0	8 BS 8 10	24 CAN 18 30	40 (28 50	56 8 38 70	72 H 48 110	88 X 58 130	104 h 68 150	120 x 78 170
1 0 0 1	9 HT 9 11	25 EM 19 31	41) 29 51	57 9 39 71	73 I 49 111	89 Y 59 131	105 i 69 151	121 y 79 171
1 0 1 0	10 LF A 12	26 SUB 1A 32	42 * 2A 52	58 : 3A 72	74 J 4A 112	90 Z 5A 132	106 j 6A 152	122 z 7A 172
1 0 1 1	11 VT B 13	27 ESC 1B 33	43 + 2B 53	59 ; 3B 73	75 K 4B 113	91 [5B 133	107 k 6B 153	123 { 7B 173
1 1 0 0	12 FF C 14	28 FS 1C 34	44 , 2C 54	60 < 3C 74	76 L 4C 114	92 \ 5C 134	108 l 6C 154	124 | 7C 174
1 1 0 1	13 CR D 15	29 GS 1D 35	45 – 2D 55	61 = 3D 75	77 M 4D 115	93] 5D 135	109 m 6D 155	125 } 7D 175
1 1 1 0	14 SO E 16	30 RS 1E 36	46 . 2E 56	62 > 3E 76	78 N 4E 116	94 ^ 5E 136	110 n 6E 156	126 ~ 7E 176
1 1 1 1	15 SI F 17	31 US 1F 37	47 / 2F 57	63 ? 3F 77	79 O 4F 117	95 _ 5F 137	111 o 6F 157	127 DEL 7F 177

LEGEND: dec / CHAR / hex oct

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

FIGURE 15.1: This lovely ASCII table shows the binary, hexadecimal, octal and decimal representations of ASCII characters (in the corners of each square; see the legend rectangle at bottom. Table produced from TeX code written and developed by Victor Eijkhout available at https://ctan.math.illinois.edu/info/ascii-chart/ascii.tex

```
# use read_lines to read the R1 fastq file line by line;
# then make a 4 column matrix, filling by rows
# then drop column 3, which corresponds to the "+" line
R1 <- read_lines("data/Battle_Creek_01_chinook_R1.fq.gz") %>%
  matrix(ncol = 4, byrow = TRUE) %>%
  .[,-3]


# add colnames
colnames(R1) <- c("ID", "seq", "qual")

# now make a tibble out that.  We will assign
# it back to the variable R1, to note carry
# extra memory around
R1 <- as_tibble(R1)

# Look at it:
R1


# OK, 1 million reads.
```

Look at the first ID line:

```
@E00430:101:HKT7WCCXY:1:1101:6411:1204 1:N:0:NGATGT
```

Aha! This is a slightly different format than the above. The part after the space has colon-separated fields that are:

```
1        :  which read of the pair
N        :  has this been filtered (Y/N)
0        :  control number (always 0 on HiSeq and NextSeq)
NGATGT   :  barcode on the read
```

OK, our mission is to actually look at the locations of these reads on different tiles. To do that we will want to access the x and y coordinates and the tiles, etc. In the tidyverse, this means giving each of those things its own column.

15.2.3.2 tidyr::separate()

How do we break those colon-separated fields into columns? This is a job for `tidyr::separate` which breaks a text string on user-defined separators into columns in a tibble.

Here we can use it to break the ID into two parts split on the space, and then break the first part of the ID into its constituent parts:

```
# first we break on the space,
# then we break the ID on the colons, but keep the original "id" for later
R1 %>%
  separate(ID, into = c("id", "part2"), sep = " ") %>%
  separate(
    id,
    into = c("machine", "run", "flow_cell", "lane", "tile", "x", "y"),
    sep = ":",
    remove = FALSE
    )
```

Wow! That was cool. Now, your mission is to pipe that (with `%>%`) into another `separate()` command that breaks part2 into `read`, `filter`, `cnum`, and `barcode`, and save that into `R1_sep`

Here is a starter:

```
R1_sep <- R1 %>%
  separate(ID, into = c("id", "part2"), sep = " ") %>%
  separate(
    id,
    into = c("machine", "run", "flow_cell", "lane", "tile", "x", "y"),
    sep = ":",
    remove = FALSE
    ) %>%
    ...your code here...

# when you are done with that, look at it
R1_sep
```

Doh! There is one thing to note. Look at the first few columns of that:

```
# A tibble: 1,000,000 x 11
   id          machine run   flow_cell lane  tile  x     y
   <chr>       <chr>   <chr> <chr>     <chr> <chr> <chr> <chr>
 1 @E00430:10… @E00430 101   HKT7WCCXY 1     1101  6411  1204
 2 @E00430:10… @E00430 101   HKT7WCCXY 1     1101  7324  1204
 3 @E00430:10… @E00430 101   HKT7WCCXY 1     1101  8582  1204
 4 @E00430:10… @E00430 101   HKT7WCCXY 1     1101  9841  1204
 5 @E00430:10… @E00430 101   HKT7WCCXY 1     1101  10186 1204
```

The x- and the y-coordinates are listed as characters (`<chr>`) but they should be numeric. This shows the default behavior of `separate()`: it just breaks each

field into a column of strings. However, you can ask `separate()` to make a good-faith guess of the type of each column and convert it to that. This works suitably in this situation, so, let's repeat the last command, but convert types automatically:

```
R1_sep <- R1 %>%
  separate(ID, into = c("id", "part2"), sep = " ") %>%
  separate(
    id,
    into = c("machine", "run", "flow_cell", "lane", "tile", "x", "y"),
    sep = ":",
    remove = FALSE,
    convert = TRUE
  ) %>%
  separate(part2, into = c("read", "filter", "cnum", "barcode"), sep = ":", convert = TRUE
```

15.2.3.3 Counting tiles

Let's see how many tiles these reads came from. Basically we just want to count the number of rows with different values for tile. Read the documentation for `dplyr::count` with

```
?count
```

and when you are done count up the number of reads in each tile:

```
R1_sep %>%
  ...your code here...
```

Turns out they are all from the same tile…

15.2.3.4 Parsing quality scores

Now, we want to turn the quality-score ASCII-characters into Phred-scaled qualities, ultimately taking the average over each sequence of those.

Check out this function:

```
utf8ToInt("!*JGH")
```

```
## [1] 33 42 74 71 72
```

We can use that to get Phred-scaled values by subtracting 33 from the result. Let's check:

```
utf8ToInt("!+5?IJ") - 33
```

```
## [1]  0 10 20 30 40 41
```

Note that `J` seems to be used for "below 1 in 10,000" as it is the highest I have ever seen.

So, what we want to do is make a new column called `mean_qual` that gives the mean of the Phred-scaled qualities. Any time you need to make a new column in a tibble, where the result in each row depends only on the values in current columns in that same row, that is a job for `mutate()`.

However, in this case, because the `utf8ToInt()` function doesn't take vector input, computing the `mean_qual` for every row requres using one of the `map()` family of functions. It is a little beyond what we want to delve into at the moment. But here is what it looks like:

```
R1_sepq <- R1_sep %>%
  mutate(mean_qual = map_dbl(.x = qual, .f = function(x) mean(utf8ToInt(x) - 33))
```

Check out the distribution of those mean quality scores:

```
ggplot(R1_sepq, aes(x = mean_qual)) +
  geom_histogram(binwidth = 1)
```

15.2.3.5 Investigating the spatial distribution of reads and quality scores

Now, use the above as a template to investigate the distribution of the x-values:

```
ggplot(R1_sepq, aes(...your code here...)) +
  geom_histogram(bins = 500)
```

And, do the same for the y-values.

```
# Dsn of y
ggplot(R1_sepq, aes(x = y)) +
  geom_histogram(bins = 500) +
  coord_flip()
```

Hmm… for fun, make a 2-D hex-bin plot

```
ggplot(R1_sepq, aes(x = x, y = y)) +
  geom_hex(bins = 100) +
  scale_fill_viridis_c()
```

That is super interesting. It looks like the flowcell or camera must have a mild issue where the smears are between y = 20,000 and 30,000.

It is natural to wonder if the quality scores of the reads that did actually get recovered from those regions have lower quality scores. This makes a hexbin plot of the mean quality scores:

```
# hexbin of the mean quality score
ggplot(R1_sepq, aes(x = x, y = y, z = mean_qual)) +
  stat_summary_hex(bins = 100) +
  scale_fill_viridis_c()
```

Cool.

15.2.4 Comparing read 1 to read 2

One question that came up in class is whether the quality of read 2 is typically lower than that of read 1. We can totally answer that with the data we have. It would involve,

1. reading in all the read 2 reads and separating columns.
2. computing the read 2 mean quality scores
3. joining (see `left_join()`) on the `id` columns and then making a scatter plot.

Go for it…

15.3 FASTA

The FASTQ format, described above, is tailored for representing short DNA sequences—and their associated quality scores—that have been generated from high-throughput sequencing machines. A simpler, leaner format is used to represent longer DNA sequences that have typically been established from

a lot of sequencing, and which no longer travel with their quality scores. This is the FASTA format, which you will typically see storing the DNA sequence from *reference genomes*. FASTA files typically use the file extensions `.fa`, `.fasta`, or `.fna`, the latter denoting it as a FASTA file of nucleotides.

In an ideal world, a reference genome would contain a single, uninterrupted sequence of DNA for every chromosome. While the resources for some well-studied species include "chromosomal-level assemblies" which have much sequence organized into chromosomes in a FASTA file, even these genome assemblies often include a large number of short fragments of sequence that are known to belong to the species, but whose location and orientation in the genome remain unknown.

More often, in conservation genetics, the reference genome for an organism you are working on might be the product of a recent, small-scale, assembly of a *low-coverage genome*. In this case, the genome may be represented by thousands, or tens of thousands, of *scaffolds*, only a few of which might be longer than one or a few megabases. All of these scaffolds go into the FASTA file of the reference genome.

Here are the first 10 lines of the FASTA holding a reference genome for Chinook salmon:

```
>CM008994.1 Oncorhynchus tshawytscha isolate JC-2011-M1
AGTGTAGTAGTATCTTACCTATATAGGGGACAGTGTAGTAGTATCTTACTTATTTGGGGGACAATGCTCTAGTGTAGTAG
AATCTTACCTTTATAGGGGACAGTGCTGGAGTGCACTGGTATCTTACCTATATAGGGGACAGTGCTGGAGTGTAGTAGTG
TCTCGGCCCACAGCCGGCAGGCCTCAGTCTTAGTTAGACTCTCCACTCCATAAGAAAGCTGGTACTCCATCTTGGACAGG
ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA
TGTGTGGTACATGTTTACAGAGAAGGAGtatattaaaaacagaaaactgTTTTGGttgaaatatttgtttttgtctgaAG
CCCGAAAAACACATGAAATTCAAAAGATAATTTGACCTACGCACTAACTAGGCTTTTCAGCAGCTCAACTACTGTCCGTT
TATTGATCTACTGTACTGCAACAACATATGTACTCACACAACAGACTATATTGGATTCAGACAGGACCTATAGGTTACCA
TGCTTCCTCTCTACAGGACCTATAGGTTACCATGCTTCCTCTCTACAAGGTCTATAGGTTACCATGCGTCCTCTCTACAG
GACCTATAGGTTACCATGCTTCCTCTCTACAGGGCCTATAGGTTACCATGCTTCCTCTCTACAGGACCTGTAGGTTACCA
```

The format is straightforward: any line starting with `>` is interpreted as a line holding the identifier of the following sequence. In this case, the identifier is `CM008994.1`. The remainder of the line (following the first white space) are further comments about the sequence, but will not typically be carried through downstream analysis (such as aligment) pipelines. In this case `CM008994.1` is the name of an assembled chromosome in this reference genome. The remaining lines give the DNA sequence of that assembled chromosome.

It is convention with FASTA files that lines of DNA sequence should be less than 80 characters, but this is inconsistently enforced by different analysis programs. However, most of the FASTA files you will see will have lines that are 80 characters long.

In the above fragment, we see DNA sequence that is either upper or lower

case. A common convention is that the lowercase bases are segments of DNA that have been inferred (by, for example RepeatMasker) to include repetitive DNA. It is worth noting this if you are trying to design assays from sequence data! However, not all reference genomes have repeat-sequences denoted in this fashion.

Most reference genomes contain gaps. Sometimes the length of these gaps can be accurately known, in which case each missing base pair is replaced by an `N`. Sometimes gaps of unknown length are represented by a string of `N`'s of some fixed length (like 100).

Finally, it is worth reiterating that the sequence in a reference-genome FASTA file represents the sequence only one strand of a double-stranded molecule. In chromosomal-scale assemblies there is a convention to use the strand that has its 5' end at the telomere of the short arm of the chromosome (Cartwright and Graur, 2011). Obviously, such a convention cannot be enforced in a low-coverage genome in thousands of pieces. In such a genome, different scaffolds will represent sequence on different strands; however the sequence in the FASTA file, whichever strand it is upon, is taken to be the reference, and that sequence is referred to as the *forward* strand of the reference genome.

15.3.1 Genomic ranges

Almost every aspect of genomics or bioinformatics involves talking about the "address" or "location" of a piece of DNA in the reference genome. These locations within a reference genome can almost universally be described in terms of genomic range with a format that looks like:

```
SegmentName:start-stop
```

For example,

```
CM008994.1:1000001-1001000
```

denotes the 1 Mb chunk of DNA starting from position 1000001 and proceeding to (and including!) position 1001000. Such nomenclature is often referred to as the *genomic coordinates* of a segment.

In most applications we will encounter, the first position in a chromosome labeled 1. This is called a *base 1 coordinate system*. In some genomic applications, a *base 0 coordinate system* is employed; however, for the most part such a system is only employed internally in the guts of code of software that we will use, while the user interface of the software consistently uses a base 1 coordinate system.

15.3.2 Extracting genomic ranges from a FASTA file

Commonly (for example, when designing primers for assays) it is necessary to pick out an precise genomic range from a reference genome. This is something that you should *never* try to do by hand. That is too slow and too error prone. Rather the software package `samtools` (which will be discussed in detail later) provides the `faidx` utility to *index* a FASTA file. It then uses that index to provide lightning fast access to specific genomic coordinates, returning them in a new FASTA file with identifiers giving the genomic ranges. Here is an example using `samtools faidx` to extract four DNA sequences of length 150 from within the Chinook salmon genome excerpted above:

```
# assume the working directory is where the fasta file resides

# create the index
samtools faidx GCA_002831465.1_CHI06_genomic.fna

# that created the file: GCA_002831465.1_CHI06_genomic.fna.fai
# which holds four columns that constitute the index

# now we extract the four sequences:
samtools faidx \
    GCA_002831465.1_CHI06_genomic.fna \
    CM009007.1:3913989-3914138 \
    CM009011.1:2392339-2392488 \
    CM009013.1:11855194-11855343 \
    CM009019.1:1760297-1760446

# the output is like so:
>CM009007.1:3913989-3914138
TTACCGAtggaacattttgaaaaacacaaCAATAAAGCCTTGTGTCCTATTGTTTGTATT
TGCTTCGTGCTGTTAATGGTAgttgcacttgattcagcagccgtAGCGCCGGGAAggcag
tgttcccattttgaaaaaTGTCATGTCTGA
>CM009011.1:2392339-2392488
gatgcctctagcactgaggatgccttagaccgctgtgccactcgggaggccttcaGCCTA
ACTCTAACTGTAAGTAAATTGTGTGTATTTTTGGGTACATTTCGCTGGTCCCCACAAGGG
GAAAGggctattttaggtttagggttaagg
>CM009013.1:11855194-11855343
TGAGGTTTCTGACTTCATTTTCATTCACAGCAGTTACTGTATGCCTCGGTCAAATTGAAA
GGAAAGTAAAGTAACCATGTGGAGCTGtatggtgtactgtactgtactgtattgtactgt
attgtgtgGGACGTGAGGCAGGTCCAGATA
>CM009019.1:1760297-1760446
ttcccagaatctctatgttaaccaaggtgtttgcaaatgtaacatcagtaggggagagag
```

```
aggaaataaaggggaagaggtatttatgactgtcataaacctacccctcaggccaacgt
catgacactcccgttaatcacacagactGG
```

15.3.3 Downloading reference genomes from NCBI

Gotta write a blurb here about how much nicer it is to use the ftp link: ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/. More on this, eventually…

15.4 Alignments

A major task in bioinformatics is *aligning* reads from a sequencing machine to a reference genome. We will discuss the operational features of that task in a later chapter, but here we treat the topic of the SAM, or Sequence Alignment Map, file format which is widely used to represent the results of sequence alignment. We attempt to motivate this topic by first considering a handful of the intricacies that arise during sequence alignment, before proceeding to a discussion of the various parts of the SAM file that are employed to handle the many and complex ways in which DNA alignments can occur and be represented. This will necessarily be an incomplete and relatively humane introduction to SAM files. For the adventurous a more complete—albeit astonishingly terse—description of the SAM format specification[1] is maintained and regularly updated.

15.4.1 How might I align to thee? Let me count the ways...

We are going to consider the alignment of very short (10 bp) paired reads from the ends of a short (50 bp) fragment from the fourth line of the FASTA file printed above. In other words, those 80 bp of the reference genome are:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
```

And we will be considering double-stranded DNA occupying the middle 50 base pairs of that piece of reference genome. That piece of double stranded DNA looks like:

```
5'  ACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGA  3'
    ||||||||||||||||||||||||||||||||||||||||||||||||||
```

[1] https://samtools.github.io/hts-specs/SAMv1.pdf

```
3'  TGGACGTCCTGTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACT  5'
```

If we print it alongside (underneath, really) our reference genome, we can see where it lines up:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
               5'  ACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGA  3'
                   ||||||||||||||||||||||||||||||||||||||||||||||||||
               3'  TGGACGTCCTGTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACT  5'
```

Now, remember that any template being sequenced on an Illumina machine is going to be single-stranded, and we have no control over which strand, from a double-stranded fragment of DNA, will get sequenced. Futhermore, recall that for this tiny example, we are assuming that the reads are only 10 bp long. Ergo, if everything has gone according to plan, we can expect to see two different possible *templates*, where I have denoted the base pairs that do not get sequenced with `-`'s

```
either:
               5'  ACCTGCAGGA----------------------------AACACAGTGA  3'
or:
               3'  TGGACGTCCT----------------------------TTGTGTCACT  5'
```

If we see the top situation, we have a situation in which the template that reached the lawn on the Illumina machine comes from the strand that is represented in the reference genome. This is called the *forward strand*. On the other hand, the bottom situation is one in which the template is from the reverse complement of the strand represented by the reference. This is called the *reverse strand*.

Now, things start to get a little more interesting, because we don't get to look at the the entire template as one contiguous piece of DNA in the 5' to 3' direction. Rather, we get to "see" one end of it by reading Read 1 in the 5' to 3' direction, and then we "see" the other end of it by reading Read 2, also in the 5' to 3' direction, *but Read 2 is read off the complementary strand*.

So, if we take the template from the top situation:

```
the original template is:
               5'  ACCTGCAGGA----------------------------AACACAGTGA  3'

So the resulting reads are:

Read 1:        5'  ACCTGCAGGA  3'   --> from 5' to 3' on the template
Read 2:        5'  TCACTGTGTT  3'   --> the reverse complement of the
                                        read on the 3' end of the template
```

And if we take the template from the bottom scenario:

```
the original template is:
```

```
              3'  TGGACGTCCT----------------------------TTGTGTCACT  5'
```

So the resulting reads are:

```
Read 1:        5'  TCACTGTGTT  3'   --> from 5' to 3' on the template
Read 2:        5'  ACCTGCAGGA  3'   --> the reverse complement of the
                                        read on the 3' end of the template
```

Aha! Regardless of which strand of DNA the original template comes from, sequences must be read off of it in a 5' to 3' direction (as that is how the biochemistry works). So, there are only two possible sequences you will see, and these correspond to reads from 5' to 3' off of each strand. So, the only difference that happens when the template is from the forward or the reverse strand (relative to the reference), is whether Read 1 is from the forward strand and Read 2 is from the reverse strand, or whether Read 1 is from the reverse strand and Read 2 is from the forward strand. The actual pair of sequences you will end up seeing is still the same.

So, to repeat, with a segment of DNA that is a faithful copy of the reference genome, there are only two read sequences that you might see, and as we will show below *Read 1 and Read 2 must align to opposite strands of the reference.*

What does a faithful segment from the reference genome look like in alignment? Well, in the top case we have:

```
      Read 1:  5'  ACCTGCAGGA  3'
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
forward-strand
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
reverse-strand
3'  TGTATCTGTCCCTGGTGGACGTCCTGTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACTTGTCGTATATGGTCT  5'
                                                Read 2:  3'  TTGTGTCACT  5'
```

And in the bottom case we have:

```
      Read 2:  5'  ACCTGCAGGA  3'
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
forward-strand
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
reverse-strand
3'  TGTATCTGTCCCTGGTGGACGTCCTGTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACTTGTCGTATATGGTCT  5'
                                                Read 1:  3'  TTGTGTCACT  5'
```

Note that, although one of the reads always aligns to the reverse strand, the position at which it is deemed to align is still read off of the position on the forward strand. (Thank goodness for that! Just think how atrocious it would be if we counted positions of fragments mapping to the reverse strand by

starting from the reverse strands's 5' end, on the other end of the chromosome, and counting forward from there!!)

Note that the *alignment position* is one of the most important pieces of information about an alignment. It gets recorded in the **POS** column of an alignment. It is recorded as the first position (counting from 1 on the 5' end of the forward reference strand) at which the alignment starts. Of course, the name of the reference sequence the read maps to is essential. In a SAM file this is called the **RNAME** or reference name.

Both of the last two alignments illustrated above involve paired end reads that align "properly," because one read in the pair aligns to the forward strand and one read aligns to the reverse strand of the reference genome. As we saw above, that is just what we would expect if the template we were sequencing is a faithful copy (apart from a few SNPs or indels) of either the forward or the reverse strand of the reference sequence. In alignment parlance we say that each of the reads is "mapped in a proper pair." This is obviously an important piece of information about an alignment and it is recorded in a SAM file in the so-called **FLAG** column for each alignment. (More on these flags later…)

How can a read pair not be properly mapped? There are a few possibilities:

(1) One read of the pair gets aligned, but the other does not. For example something that in our schematic would look like this:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
               5'  ACCTGCAGGA  3'
```

(2) Both reads of the pair map to the same strand. If our paired end reads looked like:

```
Read 1:   5'  ACCTGCAGGA  3'
Read 2:   5'  AACACAGTGA  5'
```

then they would both align nicely to just the forward strand of the reference genome:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
               5'  ACCTGCAGGA------------------------------AACACAGTGA  3'
```

And, as we saw above, this would indicate the the template must not conform to the reference genome in some way. This may occur if there is a rearrangement (like an inversion, etc.) in the genome being sequenced, relative to the reference genome.

(3) The two different reads of a pair get aligned to different chromosomes/scaffolds or they get aligned so far apart on the same chromosome/scaffold that the alignment program determines the pair to be aberrant. This evaluation requires that the program have a lot

of other paired end reads from which to estimate the distribution, in the sequencing library, of the *template length*—the length of the original template. The template length for each read pair is calculated from the mapping positions of the two reads and is stored in the **TLEN** column of the SAM file.

What is another way I might align to thee? Well, one possbility is that a read pair might align to many different places in the genome (this can happen if the reads are from a repetitive element in the genome, for example). In such cases, there is typically a "best" or "most likely" alignment, which is called the *primary alignment*. The SAM file output might record other "less good" alignments, which are called *secondary alignments* and whose status as such is recorded in the **FLAG** column. The aligner `bwa mem` has an option to allow you to output all secondary alignments. Since you don't typically output and inspect all secondary alignments (something that would be an unbearable task), most aligners provide some measure of confidence about the alignment of a read pair. The aligner, *bwa*, for example, looks at all possible alignments and computes a score for each. Then it evaluates confidence in the primary alignment by comparing its score to the sum of the scores of all the alignments. This provides the *mapping quality score* found in the **MAPQ** column of a SAM file. It can be interpreted, roughly, as the probability that the given alignment of the read pair is incorrect. These can be small probabilities, and are represented as Phred scaled values (using integers, not characters!) in the SAM file.

The last way that a read might align to a reference is by not perfectly matching every base pair in the reference. Perhaps only the first part of the read matches base pairs in the reference, or maybe the read contains an insertion or a deletion. For example, if instead of appearing like `5' ACCTGCAGGA 3'`, one of our reads had an insertion of AGA, giving: `5' ACCAGAGTGCAGGA 3'`, this fragment would still align to the reference, at 10 bp, and we might record that alignment, but would still want a compact way of denoting the position and length of the insertion—a task handled by the **CIGAR** column.

To express all these different ways in which an alignment can occur, each read occupies a single line in a SAM file. This row holds information about the read's alignment to the reference genome in a number of TAB-delimited columns. There are 11 required columns in each alignment row, after which different aligners may provide additional columns of information. Table 15.1 gives a brief description of the 11 required columns (intimations of most of which occurred in **ALL CAPS BOLDFACE** in the preceding paragraphs. Some, like **POS** are relatively self-explanatory. Others, like **FLAG** and **CIGAR** benefit from further explanation as given in the subsections below.

TABLE 15.1: Brief description of the 11 required columns in a SAM file.

Column	Field	Data Type	Description
1	QNAME	String	Name/ID of the read (from FASTQ file)
2	FLAG	Integer	The SAM flag
3	RNAME	String	Name of scaffold/chromosome the read aligns to
4	POS	Integer	1-based 5'-most alignment position on reference forward strand
5	MAPQ	Integer	Phred-scaled mapping quality score
6	CIGAR	String	String indicating matches, indels and clipping in alignment
7	RNEXT	String	Scaffold/chromosome that the read's mate aligns to
8	PNEXT	Integer	Alignment position of the read's mate
9	TLEN	Integer	Length of DNA template whose ends were read (in paired-end sequencing)
10	SEQ	String	The sequence of the read, represented in 5' to 3' on the reference forward strand
11	QUAL	String	Base quality scores, ordered from 5' to 3' on the reference forward strand

15.4.2 Play with simple alignments

Now, everyone **who has a Mac** should clone the RStudio project repository on GitHub at `https://github.com/eriqande/alignment-play` (by opening a new RStudio project with the "From Version Control" –> GitHub option, for example).

This has a notebook that will let us do simple alignments and familiarize ourselves with the output in SAM format.

15.4.3 SAM Flags

The FLAG column expresses the status of the alignment associated with a given read (and its *mate* in paired-end sequencing) in terms of a combination of 12 yes-or-no statements. The combination of all of these "yesses" and "nos" for a given aligned read is called its SAM *flag*. The yes-or-no status of any single one of the twelve statements is called a "bit" because it can be thought of as a single binary digit whose value can be 0 (No/False) or 1 (Yes/True). Sometimes a picture can be helpful: we can represent each statement as a circle which is shaded if it is true and open if it is false. Thus, if all 12 statements are false you would see ○○○○ ○○○○ ○○○○. However, if statements 1, 2, 5, and 7 are true then you would see ○○○○ ○●○● ○○●●. In computer parlance we would say that bits 1, 3, 5, and 7 are "set" if they indicate Yes/True. As these are bits in a binary number, each bit is associated with a power of 2 as shown in Table 15.2, which also lists the meaning of each bit.

TABLE 15.2: SAM flag bits in a nutshell. The description of these in the SAM specification is more general, but if we restrict ourselves to paired-end Illumina data, each bit can be interpreted by the meanings shown here. The "bit-grams" show a visual representation of each bit with open circles meaning 0 or False and filled circles denoting 1 or True. The bit grams are broken into three groups of four, which show the values that correspond to different place-columns in the hexadecimal representation of the bit masks.

bit-#	bit-gram	2^x	dec	hex	Meaning
1	○○○○ ○○○○ ○○○●	2^0	1	0x1	the read is paired (i.e. comes from paired-end sequencing.)
2	○○○○ ○○○○ ○○●○	2^1	2	0x2	the read is mapped in a proper pair
3	○○○○ ○○○○ ○●○○	2^2	4	0x4	the read is not mapped/aligned
4	○○○○ ○○○○ ●○○○	2^3	8	0x8	the read's mate is not mapped/aligned
5	○○○○ ○○○● ○○○○	2^4	16	0x10	the read maps to the reverse strand
6	○○○○ ○○●○ ○○○○	2^5	32	0x20	the read's mate maps to the reverse strand
7	○○○○ ○●○○ ○○○○	2^6	64	0x40	the read is read 1
8	○○○○ ●○○○ ○○○○	2^7	128	0x80	the read is read 2
9	○○○● ○○○○ ○○○○	2^8	256	0x100	the alignment is not primary (don't use it!)
10	○○●○ ○○○○ ○○○○	2^9	512	0x200	the read did not pass platform quality checks

bit-#	bit-gram	2^x	dec	hex	Meaning
11	○●○○ ○○○○ ○○○○	2^{10}	1024	0x400	the read is a PCR (or optical) duplicate
12	●○○○ ○○○○ ○○○○	2^{11}	2048	0x800	the alignment is part of a chimeric alignment

If we think of the 12 bits as coming in three groups of four we can easily represent them as hexadecimal numbers. Hexadecimal number are numbers in base-16. They are expressed with a leading "0x" but otherwise behave like decimal numbers, except that instead of a 1's place, 10's place, and 100's place, and so on, we have a 1's place, a 16's place, and 256's place, and so forth. In the first group of four bits (reading from right to left) the bits correspond to 0x1, 0x2, 0x4, and 0x8, in hexadecimal. The next set of four bits correspond to 0x10, 0x20, 0x40, and 0x80, and the last set of four bits correspond to 0x100, 0x200, 0x400, 0x800. It can be worthwhile becoming comfortable with these hexadecimal names of each bit.

In the SAM format, the **FLAG** field records the decimal (integer) equivalent of the binary number that represents the yes-or-no answers to the 12 different statements. It is relatively easy to do arithmetic with the hexadecimal flags to find the decimal equivalent: add up the numbers in each of the three hexadecimal value places (the 1's, 16's, and 256's places) and multiply the result by 16 raised to the number of zeros right of the "x" in the hexadecimal number. For example if the bits set on an alignment are 0x1 & 0x2 & 0x10 & 0x40, then they sum column-wise to 0x3, and 0x50, so the value listed in the FLAG field of a SAM file would be $3 + 5 \cdot 16 = 83$.

While it is probably possible to get good at computing these 12-bit combinations from hexadecimal in your head, it is also quite convenient to use the Broad Institute's wonderful SAM flag calculator[2].

We will leave our discussion of the various SAM flag values by noting that the large SAM-flag bits (0x100, 0x200, 0x400, and 0x800) all signify something "not good" about the alignment. The same goes for 0x4 and 0x8. On the other hand, when you are dealing with paired-end data, one of the reads has to be read 1 and the other read 2, and that is known from their read names and the FASTQ file that they are in. So, we expect that 0x40 and 0x80 should always be set, trivially. With paired-end data, we are always comforted to see bits 0x1 and 0x2 set, as departures from that condition indicate that the pairing of the read alignments does not make sense given the sequence in the reference genome. As we saw in our discussion of how a template can properly map to a reference, you should be able to convince yourself that, in a properly mapped alignment, exactly one of the two bits 0x10 and 0x20 should be set for one

[2] https://broadinstitute.github.io/picard/explain-flags.html

read in the pair, and the other should be set for the other. Therefore, in good, happy, properly paired reads, from a typical whole genome sequencing library preparation, we should find either:

```
read 1 : 0x1 & 0x2 & 0x10 & 0x40 = 83
read 2 : 0x1 & 0x2 & 0x20 & 0x80 = 163
```

or

```
read 1 : 0x1 & 0x2 & 0x20 & 0x40 = 99
read 2 : 0x1 & 0x2 & 0x10 & 0x80 = 147
```

So, now that we know all about SAM flags and the values that they take, what should we do with them? First, investigating the distribution of SAM flags is an important way of assessing the nature and reliability of the alignments you have made (this is what *samtools flagstat* is for, as discussed in a later chapter). Second, you might wonder if you should do some sort of filtering of your alignments before you do variant calling. With most modern variant callers, the answer to that is, "No." Modern variant callers take account of the information in the SAM flags to weight information from different alignments, so, leaving bad alignments in your SAM file should not have a large effect on the final results. Furthermore, filtering out your data might make it hard to follow up on interesting patterns in your data, for example, the occurrence of improperly aligning reads can be used to infer the presence of inversions. If all those improperly paired reads had been discarded, they could not be used in such an endeavor.

15.4.4 The CIGAR string

CIGAR is an acronym for Compressed Idiosyncratic Gapped Alignment Report. It provides a space-economical way of describing the manner in which a single read aligns to a reference genome. It is particularly important for recording the presence of insertions or deletions within the read, relative to the reference genome. This is done by counting up, along the alignment, the number of base pairs that: *match* (`M`) the reference; that are *inserted* (`I`) into the read and absent from the reference; and that are *deleted* (`D`) from the read, but present in the reference. To arrive at the syntax of the CIGAR string you catenate a series of Number-Letter pairs that describe the sequence of matches, insertions and deletions that describe an alignment.

Some examples are in order. We return to our 80 base-pair reference from above and consider the alignment to it of a 10 bp read that looks like `5' ACCTGCAGGA 3'`:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
              5'  ACCTGCAGGA  3'
```

Such an alignment has no insertions or deletions, or other weird things, going on. So its CIGAR string would be `10M`, signifying 10 matching base pairs. A *very important* thing to note about this is that the `M` refers to bases that match in *position* in the alignment *even though they might not match the specific nucleotide types*. For example, even if bases 3 and 5 in the read don't match the exact base nucleotides in the alignment, like this:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
              5'  ACTTACAGGA  3'
```

its CIGAR string will typically still be `10M`. (The SAM format allows for an `X` to denote mismatches in the base nucleotides between a reference and a read, but I have never seen it used in practice.)

Now, on the other hand, if our read carried a deletion of bases 3 and 4. It would look like `5'  ACGCAGGA  3'` and we might represent it in an alignment like:

```
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
              5'  AC--GCAGGA  3'
```

where the `-`'s have replaced the two deleted bases. The CIGAR string for this alignment would be `2M2D6M`.

Continuing to add onto this example, suppose that not only have bases 3 and 4 been deleted, but also a four-base insertion of `ACGT` occurs in the read between positions 8 and 9 (of the original read). That would appear like:

```
5'  ACATAGACAGGGACCACCTGCAG----GACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA
              5'  AC--GCAGACGTGA  3'
```

where `-`'s have been added to the reference at the position of the insertion in the read. The CIGAR string for this arrangement would be `2M2D4M4I2M` which can be hard to parse, visually, if your eyes are getting as old as mine, but it translates to:

```
2 bp Match
2 bp Deletion
4 bp Match
4 bp Insert
2 bp Match
```

In addition to `M`, `D` and `I` (and `X`) there are also `S` and `H`, which are typically seen with longer sequences. They refer to *soft-* and *hard-clipping*, respectively, which are situations in which a terminal piece of the read, from either near the 3' or 5' end, does not align to the reference, but the central part, or the other end of the read does. Hard clipping removes the clipped sequence from the read as represented in the SEQ column, while soft clipping does not remove the clipped sequence from the SEQ column representation.

One important thing to understand about CIGAR strings is that they always represent the alignment as it appears in the 5' to 3' direction. As a consequence, it is the same whether you are reading it off the read in the 5' to 3' direction *or* if you are reading it off from how the reverse complement of the read would align to the opposite strand of the reference. Another picture is in order: if we saw a situation like the following, with a deletion in Read 1 (which aligns to the reverse strand), the CIGAR string would be, from 5' to 3' on Read 1, `6M2D2M`, which is just what we would have if we were to align the reverse complement of Read 1, called `Comp R1` below to the forward strand of the reference.

```
      Read 2:  5'  ACCTGCAGGA  3'            Comp R1:  5'  AA--CAGTGA  3'
5'  ACATAGACAGGGACCACCTGCAGGACACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3'
forward-strand
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
reverse-strand
3'  TGTATCTGTCCCTGGTGGACGTCCTGTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACTTGTCGTATATGGTCT  5'
                                            Read 1:  3'  TT--GTCACT  5'
```

For the most part, it is important to have an understanding of CIGAR strings, though you will rarely end up parsing and using them yourself. That job is best left for the specialized tools that process SAM files and their compressed equivalents, BAM files. Nonetheless, it is worth pointing out that if you want to identify the nucleotide values (i.e. alleles) at different variant positions upon single reads, it is necessary to contend with CIGAR strings to do so. This is one of the things that gets taken care of (with some Perl code) in the R package microhaplot[3] for extracting microhaplotypes from short read data (and then visualizing them).

15.4.5 The SEQ and QUAL columns

These columns hold the actual reads and quality scores that came off the sequencing machine and were in the FASTQ files. (Note, if you thought that after aligning your reads, the SAM or BAM files would end up taking up less space then the ridiculously large, gzipped FASTQ files you just downloaded from the sequencing center, guess again! SAM files actually have all the information present in a FASTQ, along with extra information about the alignments.)

The only thing that is tricky about these columns is that, if the read aligns to the reverse strand of the reference genome, the entry in the **SEQ** column is the reverse complement of the read that actually appeared in the FASTQ file. Of course, when you read off the letters of DNA from a reverse complement, going left to right the way you read a book, the order in which you encounter

[3] https://cran.r-project.org/web/packages/microhaplot/index.html

the complement of each base from the original sequence is reversed from the way you would read the bases in the original sequence. Accordingly, the order of the base quality scores that appear in the **QUAL** column will be in reverse order is the mapping was to the reverse strand.

15.4.6 SAM File Headers

To this point, we have talked almost exclusively about the rows in a SAM file which record the alignments of different reads. However, if you look at a SAM file (with `cat` or `less`, or in a text editor) the first thing you will see is the SAM file *header*: a series of lines that all start with the `@` symbol followed by two capital letters. Like `@SQ`.

These file header lines can appear daunting at first, and, when merging or dividing SAM and BAM files can prove to the bain of your existence, so understanding both their purpose and structure is paramount to avoiding some pain down the road.

In a nutshell, the lines in the header provide information to programs (and people) that will be processing the data in a SAM (or BAM, see below) file. Some of the header lines give information about the reference genome that was aligned to, others provide an overview of the various samples (and other information) included in the file, while still others give information about the program and data that produced the SAM file.

If you have every looked at the SAM header for an alignment to a reference genome that is in thousands of pieces, you were likely overwhelmed by thousands of lines that started with `@SQ`. Each of these lines gives information about the names and lengths (and optionally, some other information) of sequences that were used as the reference for the alignment.

Each header line begins with an `@` followed by two capital letters, for example, `@RG` or `@SQ`. The two-letter code indicates what kind of header line it is. There are only five kinds of header lines:

- `@HD`: the "main" header line, which, if present, will be the first line of the file. It's purpose is to reveal the version of the SAM format in use, and also information about how the alignments in the file are sorted.
- `@SQ`: typically the most abundant SAM header lines, each of these gives information about a sequence in the reference genome aligned to.
- `@RG`: these indicate information about *read groups*, which are collections of reads that, for the purposes of downstream analysis can all be assumed to be from the same individual and to have been treated the same way during the processes of library preparation and sequencing. We will discuss read groups more fully in the section on alignment.

- `@PG`: a line that tells about the program that was used to produced the SAM file.
- `@CO`: a line in which a comment can be placed.

Those first three characters signify that a line is a header line, but, within each header line, how is information conveyed? In all cases (except for the comment lines using `@CO`), information within a header line is provided in TAB-delimited, colon-separate *key-value* pairs. This means that the type or meaning or each piece of information is tagged with a key, like `ID`, and its *value* follows that key after a colon. For example, then line

```
@PG ID:bwa  PN:bwa  VN:0.7.9a-r786
```

tells us that the ID of the program that produced the SAM file was `bwa`, and its version number (VN) was 0.7.9a-r786.

The keys in the key-value pairs are always two uppercase letters. And different keys are allowed (and in some case required) in the context of each of the five different kinds of SAM header lines.

The most important keys for the different kinds of header lines are as follows:

- For `@HD`
  - `VN`: the version of the SAM specification in use. This is required.
  - `SO`: the sort order of the file. The default value is be `unknown`. More commonly, when your SAM/BAM file is prepared for variant calling, it will have been sorted in the order of the reference genome, which is denoted as `SO:coordinate`. For other purposes, it is important to sort the file in the order of read names, or `SO:queryname`. It is important to note that setting these values in a SAM file does not sort it. Rather, when you have asked a program to sort a SAM/BAM file, that program will make a note in the header about how it is sorted.
- For `@SQ`
  - `SN` is the key for the sequence name and is required.
  - `LN` is the key for the length (in nucleotide bases) of the sequence, and is also required.
- For `@RG`
  - `ID` is the only required key for `@RG` lines. Note that if multiple `@RG` lines occur in the file, each of their `ID` values must be different.
  - `SM` is the key for the name of the sample from which the reads came from. This is the name used when recording genotypes of individuals when doing variant calling.
  - `LB` denotes the particularly library in which the sample was prepared for sequencing.
  - `PU` denotes the "Platform Unit," of sequencing. Typically interpreted to mean the flow-cell and the lane upon which the sample was sequenced.

We will talk much more about the contents of read-group header lines, and how to fill them.

- For `@PG`
  - `ID` is required and provides information about the program that produced the SAM output.
  - `VN` as we saw before, this key lets you record the version of the program used to produce the SAM output.

A complete accounting of the different possible SAM header lines and their contents is given in the SAM specification[4]. It is given in a terse table that is quite informative and is not terribly tough sledding. It is recommended that you read the section on header lines.

15.4.7 The BAM format

As you might have inferred from the foregoing, SAM files can end up being enormous text files. The two big problems with that are:

1. They could take up a lot of hard drive space.
2. It would take you (or some program that was processing a SAM file) a lot of time to "scroll" through the file to find any particular alignment you (it) might be interested in.

The originators of the SAM format dealt with this by also specifying and creating a compressed *binary* (meaning "not composed of text") format to store all the information in a SAM file. This is called a BAM (Binary Alignment Map) file. In a BAM file, each column of information is stored in its native data type (i.e., the way a computer would reprensent it internally if it were working on it) and then the file holding all of these "rows" is compressed into a series of small blocks in such a way that the file can be indexed, allowing rapid access (without "scrolling" through the whole file) to the alignments within a desired genomic range. As we will see in a later chapter, in order to index such a file for rapid access to alignments in a particular genomic range, the alignments must be sorted in the order of genomic coordinates in the reference sequence.

Since BAM files are smaller than SAM files, and access into them is faster than for SAM files, you will almost always convert your SAM files to BAM files to prepare for futher bioinformatic processing. The main tool used for this purpose is the program `samtools` (written by the creators of the SAM and BAM formats) for that purpose. We will encounter `samtools` in a later chapter.

Finally, humans cannot directly read BAM files, or even decompress them with

[4] https://samtools.github.io/hts-specs/SAMv1.pdf

standard Unix tools. If you want to view a BAM file, you can use `samtools view` to read it in SAM format.

15.4.8 Quick self study

1. Suppose a read is Read 2 from the FASTQ, it aligns to the reverse strand, and its mate does too. The alignment is a primary alignment, but it has been flagged as a PCR duplicate. Write down, in hexadecimal form, all the bits that will be set for this read's alignment, then combine them to compute the SAM FLAG for it.
2. Given the alignments shown below, what do you think the CIGAR strings for Read 1 and Read 2 might look like?

```
      Read 2:  5'  ACCT--AGGAGGACACACAC   3'
5'  ACATAGACAGGGACCACCTGCAGGA---CACACACGCAGGTTTACTAAGGGTTTACTCAACACAGTGAACAGCATATACCAGA  3
forward-strand
    |||||||||||||||||||||||---||||||||||||||||||||||||||||||||||||||||||||||||||||||||
reverse-strand
3'  TGTATCTGTCCCTGGTGGACGTCCT---GTGTGTGCGTCCAAATGATTCCCAAATGAGTTGTGTCACTTGTCGTATATGGTCT  5
                                        Read 1:  3' AAAAAAAAAAAATTGTGTC-CT  5'
```

15.5 Variants

In terms of the format/standard, is going to be well worth explaining the early part of section 5 of the standard so that people know how insertions and deletions are coded. I hadn't really digested that until just today. Basically, the position in the VCF file correponds to the first character in either the REF or the ALT field.
When you remember that, it all falls into place.

VCF. I've mostly used vcftools until now, but I've gotta admit that the interface is awful with all the –recode BS. Also, it is viciously slow. So, let's just skip it all together and learn how to use bcftools. One nice thing about bcftools is that it works a whole lot like samtools, syntactically.

Note that for a lot of the commands you need to have an indexed vcf.gz.

15.6 Segments

BED

15.7 Conversion/Extractions between different formats

- vcflib's vcf2fasta takes a phased VCF file and a fasta file and spits out sequence.

15.8 Visualization of Genomic Data

Many humans, by dint of our evolution, are exceptionally visual creatures. Being able to create visual maps for different concepts is also central facilitating understanding of those concepts and extending their utility to other realms. The same is true for bioinformatic data: being able to visualize bioinformatic data can help us understand it better and to see connections between parts of our data sets that we did not, before.

The text-based bioinformatic formats we have discussed so far do not, standing by themselves, offer a rich visual experience (have you ever watched a million lines of a SAM file traverse your terminal, and gotten much understanding from that?). However, sequence data, once it has been aligned to a reference genome has an "address" or "genomic coordinates" that, by analogy to street addresses and geographic coordinates suggests that aligned sequence data might be visualized like geographic data in beautiful and/or thought-provoking "maps."

There are a handful of programs that do just that: they make compelling, interactive pictures of bioinformatic data. One of those programs (that I am partial to), called IGV (for Integative Genomics Viewer), was developed in the cross-platform language, Java, by researchers at the Broad Institute. It is available for free download from `https://software.broadinstitute.org/software/igv/`, and it should run on almost any operating system. It has been well-optimized to portray genomics data at various scales and to render an incredible amount of information in visual displays as well as text-based

"tool-tip" reports that may be activated by mousing over different parts of the display.

There is extensive documentation that goes with IGV, but it is valuable (and more fun!) to just crack open some genomic data and start playing with it. To do so, there are just a few things that you need to know:

1. The placement of all data relies on the reference genome as a sort of "base map." The reference genome serves the purpose of a latitude-longitude coordinate system that lets you make sense of spatial data on maps. Therefore, it is required for all forays with IGV. You must specify a reference genome by choosing one of the options from the "Genomes" menu. If you are working on a non-model organism of conservation concern it is likely that you will have the reference genome for that critter on your local computer, so you would "Genomes->Load Genome From File…", to show IGV where the FASTA file (cannot be compressed) of your genome is on your hard drive. Let's repeat that: if you are loading the FASTA for a reference genome into IGV, you *must* use the "Genomes" menu option. If you use "**File**->Load From File…" to try to load the reference genome, it won't let you. Don't do it! Use "**Genomes->Load Genome From File**"
2. Once your reference genome is known to IGV, you can add data from the bioinformatic formats described in this chapter *that include positions from a reference genome*. These include SAM or BAM files and VCF files (but note FASTQ files). To include data from these formats, choose "File->Load From File…". (Note, BAMs are best sorted and indexed). The data from each file that you "Load" in this manner appears in a separate *track* of data in a horizontally tiled window that is keyed to the reference genome coordinates.
3. You can zoom in and out as appropriate (and in several different ways).
4. Right clicking within any track gives a set of options appropriate for the type of track it is. For example, if you are viewing a BAM file, you can choose whether the reads joined together with their mates ("View as pairs") or not, or whether the alignments should be viewed at "full scale" ("Expanded"), somewhat mashed down ("Collapsed") or completely squashed down and small ("Squished").

15.8.1 Sample Data

About 0.5 Gb of sample data can be downloaded from https://drive.google.com/file/d/1TMug-PjuL7FYrXRTpNikAgZ-ElrvVneH/view?usp=sharing.

This download includes a zip-compressed folder called `tiny-genomic-data`. Within that folder are two more folders:

- `chinook-wgs-3-Mb-on-chr-32`: FASTQs, BAMs, and VCFs from whole genome sequencing data along a 3 Mb span of Chinook salmon chromosome 32 (which in the reference is named `NC_037124.1`). The genomic region from which the data comes from is `NC_037124.1:4,000,000-7,000,000`. The BAM and VCF files can be viewed against the reference genome in IGV.
- `mykiss-rad-and-wgs-3-Mb-chr-omy28` includes BAMs from RAD-seq data (sequenced in the study by (Prince et al., 2017)) from multiple steelhead trout individuals merged together into two BAM files. The genomic region included in those BAM files is omy28:11,200,000-12,200,000. Also included is a VCF file from whole genome resequencing data from 125 steelhead and rainbow trout in the 3 Mb region from 'omy28:10,150,000-13,150,000.

Both of the above directories include a `genome` directory that holds the FASTA that you must point IGV to. Note that in neither case does the FASTA hold the complete genome of the organism. I have merely included three chromosomes in each—the chromosome upon which the BAM and VCF data are located, and the chromosome on either side of that chromosome.

Explore these data and have fun. Some things to play with (remember to right-click [cntrl-click on a Mac] each track for a menu) :

Start with the Chinook data:

1. Load the FASTA for each data set first
2. Load a BAM file after that
3. Then load a VCF file.
4. You will likely have to zoom pretty far into a genomic region with data (see above!) before you see anything interesting.
5. Try zooming in as far as you can.
6. Toggle "View as Pairs" and see the result.
7. Play with "Collapsed/Expanded/Squished"
8. Experiment with grouping/sorting/coloring alignments by different properties
9. Use coloring to quickly find alignments with
   - `F1R2` of `F2R1` orientation
   - Insert size > 1000 bp
10. Sort by mapping quality and find some reads with MAPQ < 60

11. Zoom out and use the VCF to find a region with a high variant density, then zoom back in and view the alignments there? What do you notice about the number of reads aligning to those areas?

For the steelhead data additionally:

1. Do you see where the RAD cutsite must have been and how paired-end sequencing works from either side of the cutsite?
2. What do you notice about the orientation of Read 1 and Read 2 on either side of the cutsite?
3. Why do we see the read depth patterns we see on either side of the cutsite? (i.e., in many cases it goes up as you move away from the cutsite, and then drops off again.)
4. Do you appreciate this visual representation of how sparse RAD data is compared to whole genome resequencing data?

16

Genome Assembly

This is going to be a pretty light coverage of how it works. Maybe I could get Rachael Bay or CH to write it.

17

Alignment of sequence data to a reference genome (and associated steps)

A light treatment of how bwa works. I think I will focus solely on bwa, unless someone can convince me that there are cases where something like bowtie works better.

17.1 Preprocess ?

Will have a bit about sequence pre-processing (with WGS data it already comes demultiplexed, so maybe we can hold off on this until we get to RAD data). No, we need to talk about trimming and maybe slicing. Perhaps put that in a separate chapter of "preliminaries"

17.2 Read Groups

Gotta talk about this and make it relevant to conservation. i.e. maybe one individual was sampled with blood and also with tissue and each of those were included in four different library preps, etc. Give an example that makes it clear how it works.

17.3 Quick notes to self on chaining things:

When using samtools markdup you can do like this:

```
# first step
bwa mem chinook-play/chinook-genome-idx/GCA_002872995.1_Otsh_v1.0_genomic.fna.gz chr32-160
    samtools view -b -1 - | \
    samtools fixmate -m -O BAM - fixed.bam

# unfortunately, fixmate can't write to stdout.
# then, after this, we have to sort it into coordinate order and
# markdup them.
samtools sort fixed.bam | samtools markdup - marked.bam

# after that, fixed.bam could get tossed.
```

17.4 Merging BAM files

There is a lot of discussion on biostars about how samtools does not reconstruct the `@RG` dictionary. But I think that this must be a problem with an older version. The newer version works just fine. That said, Picard's MergeSamFiles seems to be just about as fast (in fact, faster. For a comparably sized file it took 25 minutes, and gives informative output telling you where it is at). And samtools merge is at well over 30 and only about 3/4 of the way through. Ultimately it took 37 minutes. Might have been on a slower machine…

However, if you have sliced your fastqs and mapped each separately, then samtools let's you not alter duplicate read group IDs, and so you can merge those all together faithfully, as I did in the impute project. Cool.

17.5 Divide and Conquer Strategies

At the end of each of these chapters, I think I will have a special section talking about ways that things can be divided up so that you can do it quickly, or at least, within time limits on your cluster.

18

Variant calling with GATK

Standard stuff here.

Big focus on parallelizing.

19

Bioinformatics for RAD seq data with and without a reference genome

We've gotta get our hands dirty with RAD and STACKS.

For some applications (like massive salamander genomes) this is the only way forward, at the moment. Can be useful, but is also fraught with peril.

Discuss all the problems, and strategies for dealing with them.

Note that stacks2 is a lot better than things were before. And, you can use a reference genome with them.

Amanda Stahlke gave a nice practical example at ConGen. Found all the materials on box through the link that Brian Hand gave me (in my email.)

That will be worth running through closely and carefully. Note that stacks2's denovo_map.pl script now takes all the tsv output and puts it into a series of bams.

20

Processing amplicon sequencing data

Super high read depths can cause problems for some pipelines.

I am going to mostly focus on short amplicons that are less than the number of sequencing cycles, and how we create microhaps out of those, and the great methods we have for visualizing and curating those.

21

Genome Annotation

I don't intend this to be a treatise on how to actually annotate a genome. Presumably, that is a task that involves feeding a genome and a lot of mRNA transcripts into a pipeline that then makes gene models, etc. I guess I could talk a little about that process, 'cuz it would be fun to learn more about it.

However, I will be more interested in understanding what annotation data look like (i.e. in a GFF file) and how to associate it with SNP data (i.e. using snpEff).

The GFF format is a distinctly hierarchical format, but it is still tabular, it is not in XML, thank god! 'cuz it is much easier to parse in tabular format.

You can fiddle it with bedtools.

Here is an idea for a fun thing for me to do: Take a big chunk of chinook GFF (and maybe a few other species), and then figure out who the parents are of each of the rows, and then make a graph (with dot) showing all the different links (i.e. gene -> mRNA -> exon -> CDS) etc, and count up the number of occurrences of each, in order to get a sense of what sorts of hierarchies a typical GFF file contains.

22

Whole genome alignment strategies

Basically want to talk about situations

22.1 Mapping of scaffolds to a closely related genome

I basically want to get my head fully around how SatsumaSynteny works.

After that, we might as well talk about how to get in and modify a VCF file to reflect the new positions and such. (It seems we could even add something to the INFO field that listed its position in the old scaffold system. awk + vcftools sort seems like it might be the hot ticket.)

22.2 Obtaining Ancestral States from an Outgroup Genome

For many analyses it is helpful (or even necessary) to have a guess at the ancestral state of each DNA base in a sequence. These ancestral states are often guessed to be the state of a closely related (but outgroup) species. The idea there is that it is rare for the same nucleotide to experience a substitution (or mutation) in each species, so the base carried by the outgroup is assumed to be the ancestral sequence.

So, that is pretty straightforward conceptually, but there is plenty of hardship along the way to do this. There are two main problems:

1. Aligning the outgroup genome (as a query) to the target genome. This typically produces a multiple alignment format (MAF) file. So, we have to understand that file format. (read about it here[1], on the

[1] http://genome.ucsc.edu/FAQ/FAQformat#format5

UCSC genome browser site.) A decent program to do this alignment exercise appears to be LASTZ[2]

2. Then, you might have to convert the MAF file to a fasta file to feed into something like ANGSD. It seems that Dent Earl has some tools that might to do this hhttps://github.com/dentearl/mafTools[3]. Also, the ANGSD github site has a maf2fasta[4] program, though no documentation to speak of. Or you might just go ahead and write an awk script to do it. Galaxy has a website that will do it: http://mendel.gene.cwru.edu:8080/tool_runner?tool_id=MAF_To_Fasta1, and there is an alignment too called mugsy that has a perl script associated with it that will do it: ftp://188.44.46.157/mugsy_x86-64-v1r2.3/maf2fasta.pl Note that the fasta file for ancestral sequence used by ANGSD just seems to have Ns in the places that don't have alignments.

It will be good to introduce people to those "dotplots" that show alignments.

Definitely some discussion of seeding and gap extensions, etc. The LASTZ web page has a really nice explanation of these things.

The main take home from my explorations here is that there is no way to just toss two genomes into the blender with default setting and expect that you are going to get something reasonable out of that. There is a lot of experimentation, it seems to me, and you really need to know what all the options are (this might be true of just about everything in NGS analysis, but in many cases people just use the defaults....)

22.2.1 Using LASTZ to align coho to the chinook genome

First, compile it:

```
# in: /Users/eriq/Documents/others_code/lastz-1.04.00
make
make install

# then I linked those (in bin) to myaliases
```

Refreshingly, this has almost no dependencies, and the compilation is super easy.

[2] http://www.bx.psu.edu/miller_lab/dist/README.lastz-1.02.00/README.lastz-1.02.00a.html
[3] https://github.com/dentearl/mafTools
[4] https://github.com/ANGSD/maf2fasta

Now, let's find the coho chromomsome that corresponds to omy28 on NCBI. We can get this with curl:

```
# in: /tmp
curl ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/002/021/735/GCA_002021735.1_Okis_V1/GCA_00
gunzip coho-28.fna.gz
```

Then, let's also pull that chromosome out of the chinook genome we have:

```
# in /tmp
samtools faidx  ~/Documents/UnsyncedData/Otsh_v1.0/Otsh_v1.0_genomic.fna NC_037124.1 > chi
```

Cool, now we should be able to run that:

```
time lastz chinook-28.fna coho-28.fna --notransition --step=20 --nogapped --ambiguous=iupa

real    0m14.449s
user    0m14.198s
sys 0m0.193s
```

OK, that is ridiculously fast. How about we make a file that we can plot in R?

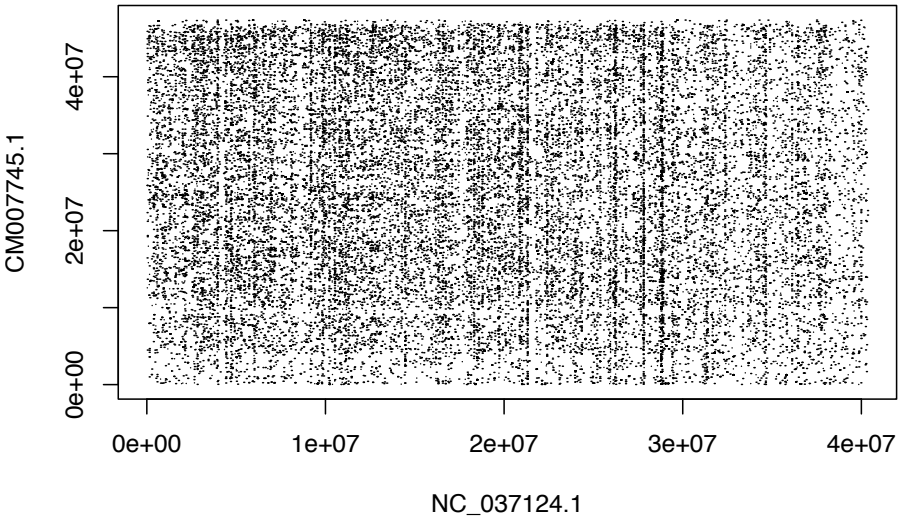
```
time lastz chinook-28.fna coho-28.fna --notransition --step=20 --nogapped --ambiguous=iupa
```

I copied that to `inputs` so we can plot it:

```
dots <- readr::read_tsv("inputs/chin28_vs_coho28.rdp.gz")
```

```
## Parsed with column specification:
## cols(
##   NC_037124.1 = col_double(),
##   CM007745.1 = col_double()
## )
```

```
plot(dots,type="l")
```



OK, clearly what we have there is just a bunch of repetitive bits. I think we must not have the same chromosomes in the two species.

So, let's put LG28 in coho against the whole chinook genome. Note the use of the bracketed "multiple" in there to let it know that there are multiple sequences in there that should get catenated together.

```
time lastz ~/Documents/UnsyncedData/Otsh_v1.0/Otsh_v1.0_genomic.fna[multiple]  coho-28.fna

FAILURE: in load_fasta_sequence for /Users/eriq/Documents/UnsyncedData/Otsh_v1.0/Otsh_v1.0
```

No love there. But that chinook genome has a lot of short scaffolds in there too, I think.

Maybe we could just try LG1. Nope. How about we toss every coho LG against LG1 from chinook…

```
# let's get the first 10 linkage groups from coho:
for i in {1..10}; do curl ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/002/021/735/GCA_00202

# now lets try aligning those to the chinook
for i in {1..10}; do time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --nog
```

Nothing looked good until I got to coho LG10:

```
dots <- readr::read_tsv("inputs/chinook1_vs_coho10.rdp.gz")

plot(dots,type="l")
```

There is clearly a big section that aligns there. But, we clearly are going to need to clean up all the repetive crap, etc on these alignments.

22.2.2 Try on the chinook chromosomes

So, it crapped out on the full Chinook fasta. Note that I could modify the code (or compile it with a `-D`): check this out in sequences.h:

```
//  Sequence lengths are normally assumed to be small enough to fit into a
//  31-bit integer.  This gives a maximum length of about 2.1 billion bp, which
//  is half the length of a (hypothetical) monoploid human genome.  The
//  programmer can override this at compile time by defining max_sequence_index
//  as 32 or 63.
```

But, for now, I think I will just go for the assembled chromosomes only:

```
# just get the well assembled chromosomes (about 1.7 Gb)
# in /tmp
samtools faidx ~/Documents/UnsyncedData/Otsh_v1.0/Otsh_v1.0_genomic.fna $(awk '/^NC/ {prin

# then try tossing coho 1 against that:
time lastz chinook_nc_chroms.fna[multiple]  coho-1.fna --notransition --step=20 --nogapped

# that took about 7 minutes
real    6m33.411s
user    6m28.391s
sys 0m4.121s
```

Here is what the figure looks like. It is clear that the bulk of Chromosome 1 in coho aligns for long distances to two different chromosomes in Chinoook (likely paralogs?). This is complex!

22.2.3 Explore the other parameters more

I am going to use the chinook chromosome 1 and coho 10 to explore things that will clean up the results a little bit.

```
# in /tmp
 i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --nogapped --ambigu
```

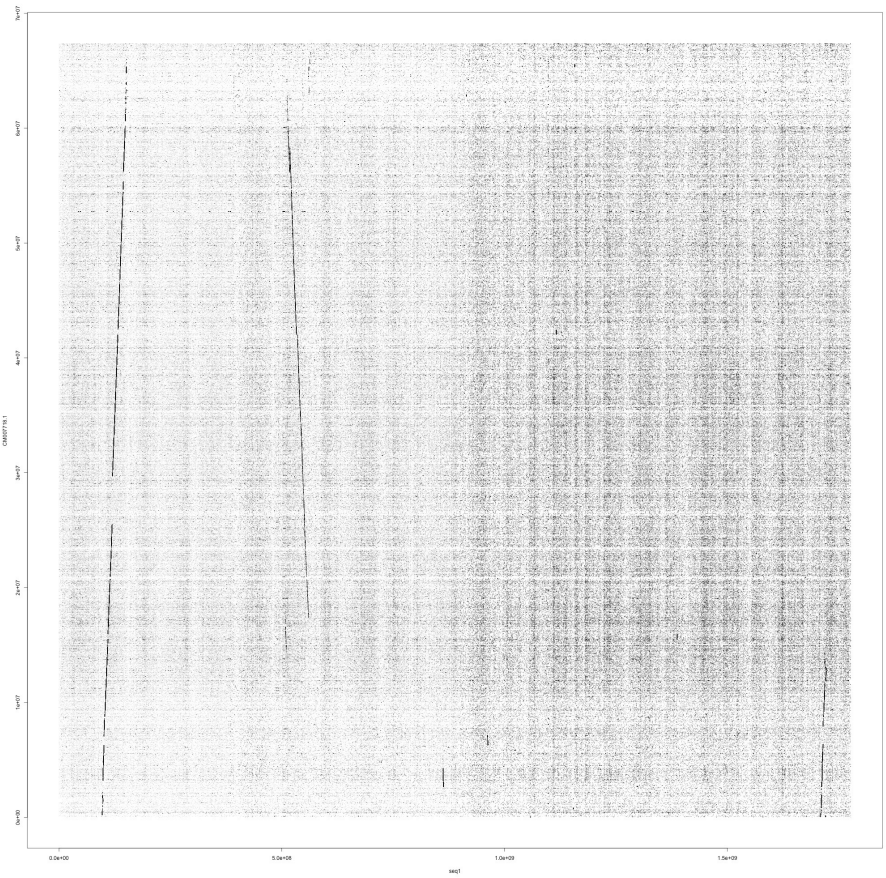


FIGURE 22.1: Coho Chromo 1 on catenated chinook chromos

```
real    0m29.055s
user    0m28.497s
sys 0m0.413s
```

That is quick enough to explore a few things. Note that we are already doing gap-free extension, because "By default seeds are extended to HSPs using x-drop extension, with entropy adjustment." However, by default, chaining is not done, and that is the key step in which a path is found through the high-scoring pairs (HSPs). That doesn't take much (any) extra time and totally cleans things up.

```
i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --nogapped --ambiguo

real    0m28.704s
```

So, that is greatly improved:

```
dots <- readr::read_tsv("inputs/chain-chinook1_vs_coho10.rdp.gz")

plot(dots,type="l")
```

So, it would be worth seeing if chaining also cleans up the multi-chrom alignment:

```
time lastz chinook_nc_chroms.fna[multiple]  coho-1.fna --notransition --step=20 --nogapped

real    6m42.835s
user    6m35.042s
sys 0m6.207s
```

```
dots <- readr::read_tsv("inputs/chin_nc_vs_coho1-chain.rdp.gz")

plot(dots,type="l")
```

OK, that shows that it will find crappy chains if given the chance. But if you zoom in on that stuff you see that some of the spots are pretty short, and some are super robust. So, we will want some further filtering to make this work. So, we need to check out the "back-end filtering." that is possible. Back-end filtering does not happen by default.

Let's say that we want 70 Kb alignment blocks. That is .001 of the total input sequence.

```
time lastz chinook_nc_chroms.fna[multiple]  coho-1.fna --notransition --step=20 --nogapped
```

That took 6.5 minutes again. But, it also produced no output whatsoever. We probably want to filter on identity first anyway. Because that takes so long, maybe we could do it with our single chromosome first.

```
i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --nogapped --ambiguo
```

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho10-ident95.rdp")

plot(dots,type="l")
```

That keeps things very clean, but the alignment blocks are all pretty short (like 50 to 300 bp long). So perhaps we need to do gapped extension here to make these things better. This turns out to take a good deal longer.

```
i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --gapped --ambiguous

real    3m15.575s
user    3m14.048s
sys 0m0.936s
```

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho10-ident95-gapped.rdp")
plot(dots,type="l")
```

That is pretty clean and slick.

Now, this has got me to thinking that maybe I *can* do this on a chromosome by chromosome basis.

Check what 97% identity looks like:

```
i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --gapped --ambiguous
```

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho10-ident97-gapped.rdp")
plot(dots,type="l")
```

That looks to have a few more holes in it.

Final test. Let's see what happens when we chain it on a chromosome that doesn't have any homology:

```
# first with no backend filtering
i=1; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --gapped --ambiguous=

real    0m35.130s
user    0m34.642s
sys 0m0.413s
```

```
# Hey! That is cool.  When there are no HSPs to chain, this doesn't take very long
i=1; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --gapped --ambiguous=
```

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho1-gapped.rdp")
plot(dots,type="l")
```

OK, it finds something nice and crappy there.

What about if we require 95% identity?

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho1-gapped-ident95.rdp")
plot(dots,type="l")
```

That leaves us with very little.

Let's also try interpolation at the end to see how that does. Note that here we also produce the rdotplot at the same time as the maf.

```
i=10; time lastz chinook-1.fna coho-${i}.fna --notransition --step=20 --gapped --ambiguous

real    4m25.625s
user    4m22.957s
sys 0m1.478s
```

That took an extra minute, but was not so bad.

```
dots <- readr::read_tsv("/tmp/chinook1_vs_coho10-ident95-gapped-inner1000.rdp")
plot(dots,type="l")
```

22.2.3.1 Repeat Masking the Coho genome

Turns out that NCBI site has the repeat masker output in GCF_002021735.1_Okis_V1_rm.out.gz. I save that to a shorter name. Now I will make a bed file of the repeat regions. Then I use bedtools maskfasta to softmask that fasta file.

```
# in: /Users/eriq/Documents/UnsyncedData/Okis_v1
gzcat Okis_V1_rm.out.gz | awk 'NR>3 {printf("%s\t%s\t%s\n", $5, $6, $7)}' > repeat-regions

bedtools maskfasta -fi Okis_V1.fna -bed repeat-regions.bed  -fo Okis_V1-soft-masked.fna  -
```

That works great. But it turns out that the coho genome is already softmasked.

But, it is good to now that I can use a repeat mask output file to toss repeat sites if I want to for ANGSD analyses, etc.

22.2.3.2 multiz maf2fasta

So, it looks like you can use single_cov2 from multiz to retain only a single alignment block covering each area. Then you can maf2fasta that and send it off in fasta format. Line2 holds the reference (target) sequence, but it has dashes added where the query has stuff not appearing in the target. So, what you have to do is run through that sequence and drop all the positions in the query that correspond to dashes in the target. That will get us what we want.

But maybe I can just use megablast like Christensen and company. They have some of their scripts, but it is not clear to me that it will be easy to get that back to a fasta for later analysis in ANGSD.

Not only that, but then there is the whole paralog issue. I am exploring that a little bit right now. It looks like when you crank the identity requirement up, the paralogs get pretty spotty so they can be easily recognized. For example setting the identity at 99.5 makes it clear which is the paralog:

```
time lastz chinook_nc_chroms.fna[multiple]  coho-1.fna --notransition --step=20 --nogapped
# takes about 6 minutes
```

And the figure is here.

So, I think this is going to be a decent workflow:

1. run lastz on each coho linkage group against each chinook chromosome separately. Do this at identity=92 and identity=95 and identity=99.9. For each run, produce a .maf file and at the same time a .rdotplot output.
2. Combine all the .rdotplot files together into something that can be faceted over chromosomes and make a massive facet grid showing the results for all chromosomes. Do this at different identity levels so that the paralogs can be distinguished.
3. Visually look up each columns of those plots and determine which coho chromosomes carry homologous material for each chinook chromosome. For each such chromosome pair, run single_cov2 on them (maybe on the ident=92 version).
4. Then merge those MAFs. Probably can just cat them together, but there might be some sorting that needs to be done on them.
5. Run maf2fasta on those merged mafs to get a fasta for each chinook chromosome.

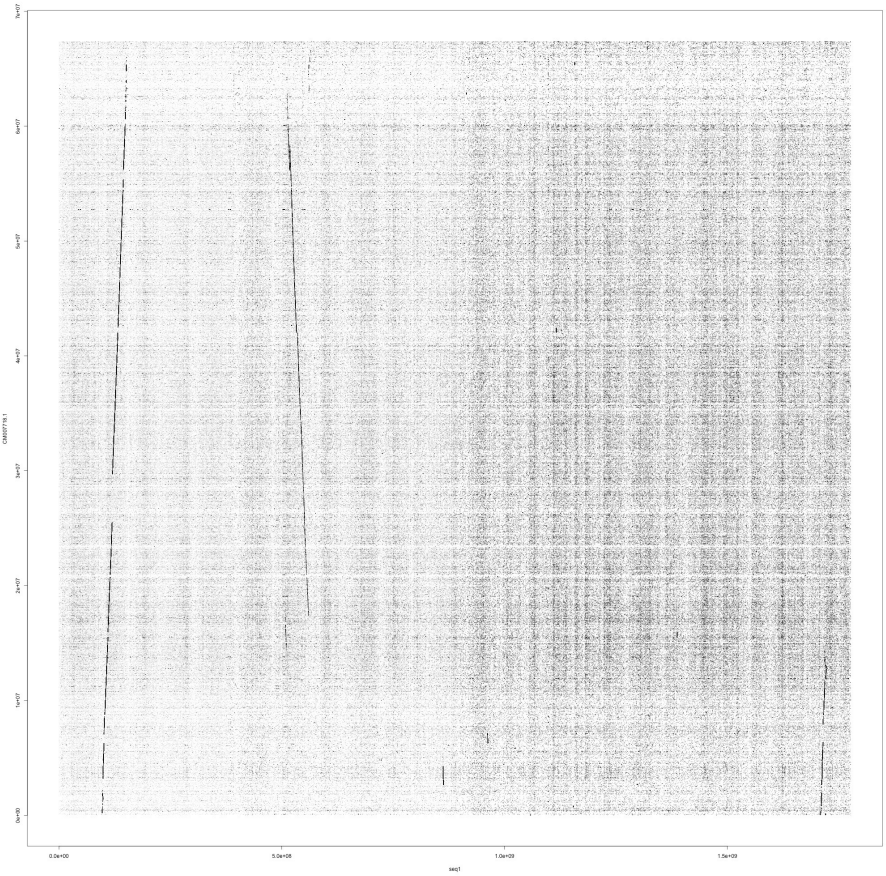


FIGURE 22.2: Coho Chromo 1 on catenated chinook chromos. Ident=99.5

6. Write a C-program that uses uthash to efficiently read the fasta for each chinook chromosome and then write out a version in which the positions that are dashes in the chinook reference are removed from both the chinook reference and the aligned coho sequence. _Actually, one can just pump each sequence out to a separate file in which each site occupies one line. Then paste those and do the comparison…

```
2018-10-18 11:15 /tmp/--% time (awk 'NR==2' splud | fold -w1 > spp1.text)

real    0m23.244s
user    0m22.537s
```

```
sys 0m0.685s

# then you can use awk easily like this:
paste spp1.text spp2.text | awk 'BEGIN {SUBSEP = " "} {n[$1,$2]++} END {for(i in n) print
```

7. The coho sequence thus obtained will have dashes anywhere there isn't coho aligned to the chinook. So, first, for each chromosome I can count the number of dashes, which will tell me the fraction of sites on the chinook genome that were aligned (sort of—there is an issue with N's in the coho genome.) Then those dashes can be converted to N's.
8. It would be good to count the number of sites that are not N's in chinook that are also not Ns in coho, to know how much of it we have aligned.

Note, the last thing that really remains here is making sure that I can run two or more different query sequences against one chinook genome and then process that out correctly into a fasta.

Note that Figure 1 in christensen actually gives me a lot to go on in terms of which chromosomes in coho to map against which ones in chinook.

Part IV

Part IV: Analysis of Big Variant Data

23

Bioinformatic analysis on variant data

Standard analyses like computing Fst and linkage disequilibrium, etc., from data, typically in a VCF file.

Basically, we want to get comfortable with plink 2.0, bedtools, vcftools, etc.

The key in all of this is to motivate every single thing we do here in terms of an application in conservation genomics. That is going to be key.

This Part IV will be about standard bioinformatic tools for doing things with big variant data.

- Filtering
- Imputation
- LD, HWD, FST
- Etc.

I will probably have a chapter on unix tools.

Maybe another on R/Bioconductor tools.

Gotta have a chapter about "Look at your data!" and Whoa! and diagnostics using radiator.

Part V

Part V: Population Genomics

24

Topics in pop gen

This is just a bunch of ideas. But basically, I want to have some topics here that everyone should know about. Slanted toward things that are relevant for inference or simulation.

24.1 Coalescent

Gotta have a lecture on the coalescent. It would be nice to try to motivate all the topics from this backward in time perspective.

Get far enough to discuss π and the expected site frequency spectrum.

24.2 Measures of genetic diversity and such

It would really be good for me to write a chapter / give a few lectures on different measures like dxy and fst

From Ash's paper: However, population genomic analyses (outlined below) use FST only, as dxy was highly correlated to nucleotide diversity (for early stage diverging populations the correlation between dxy and π is > 0.91, Pearson correlation). As such variation in dxy across the genome reflects variation in diversity, not differentiation (Riesch et al., 2017).

Tajima's D and such. The influence of selection on such measures.

24.3 Demographic inference with $\partial a \partial i$ and *moments*

24.4 Balls in Boxes

Would be worthwhile to have a review of all these sorts of variants of population assignment, structure, admixture, etc.

Population structure and PCAs.

finestructure and fineRADstructure.

Might want to insert Bradburd et al. (2018).

Might also want to discuss Pickrell and Pritchard (2012).

Also: Pritchard et al. (2000)

What if we go and try to put the same one in? Like Pritch 2000 again: (Pritchard et al., 2000)

24.5 Some landscape genetics

After talking with Amanda about her dissertation I realized it would be good to talk about some landscape genetics stuff. For sure I want to talk about EEMS and maybe CircuitScape, just so I know well what is going on with the latter.

24.6 Relationship Inference

Maybe do a lecture on this…

24.7 Tests for Selection

A look at a selection of the methods that are out there. FST outliers, *Bayescan*, *Lositan*, *PCAdapt*, and friends. It would be good to get a nice succinct explanation/understanding of all of these.

24.8 Multivariate Associations, GEA, etc.

It really is time for me to wrap my head around this stuff.

24.9 Estimating heritability in the wild

Another from Amanda. It would be good to do some light Quant Genet so that we all understand how we might be able to use NGS data to estimate heritability in wild populations.

Bibliography

Barson, N. J., Aykanat, T., Hindar, K., Baranski, M., Bolstad, G. H., Fiske, P., Jacq, C., Jensen, A. J., Johnston, S. E., Karlsson, S., Kent, M., Moen, T., Niemelä, E., Nome, T., Næsje, T. F., Orell, P., Romakkaniemi, A., Sægrov, H., Urdal, K., Erkinaro, J., Lien, S., and Primmer, C. R. (2015). Sex-dependent dominance at a single locus maintains variation in age at maturity in salmon. *Nature*, 528(7582):405–408.

Bradburd, G. S., Coop, G. M., and Ralph, P. L. (2018). Inferring Continuous and Discrete Population Genetic Structure Across Space. *Genetics*, 210(1):33–52.

Cartwright, R. A. and Graur, D. (2011). The multiple personalities of Watson and Crick strands. *Biology Direct*, 6(1):7.

Pickrell, J. K. and Pritchard, J. K. (2012). Inference of Population Splits and Mixtures from Genome-Wide Allele Frequency Data. *PLOS Genetics*, 8(11):e1002967.

Prince, D. J., O'Rourke, S. M., Thompson, T. Q., Ali, O. A., Lyman, H. S., Saglam, I. K., Hotaling, T. J., Spidle, A. P., and Miller, M. R. (2017). The evolutionary basis of premature migration in Pacific salmon highlights the utility of genomics for informing conservation. *Science Advances*, 3(8):e1603198.

Pritchard, J. K., Stephens, M., and Donnelly, P. (2000). Inference of Population Structure Using Multilocus Genotype Data. *Genetics*, 155(2):945–959.

Raymond, E. S. (2003). *Art of UNIX Programming, The*. Addison-Wesley Professional. Part of the Addison-Wesley Professional Computing Series series., 1st edition.

Watson, J. D. and Crick, F. H. C. (1953). Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature*, 171(4356):737–738.