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.

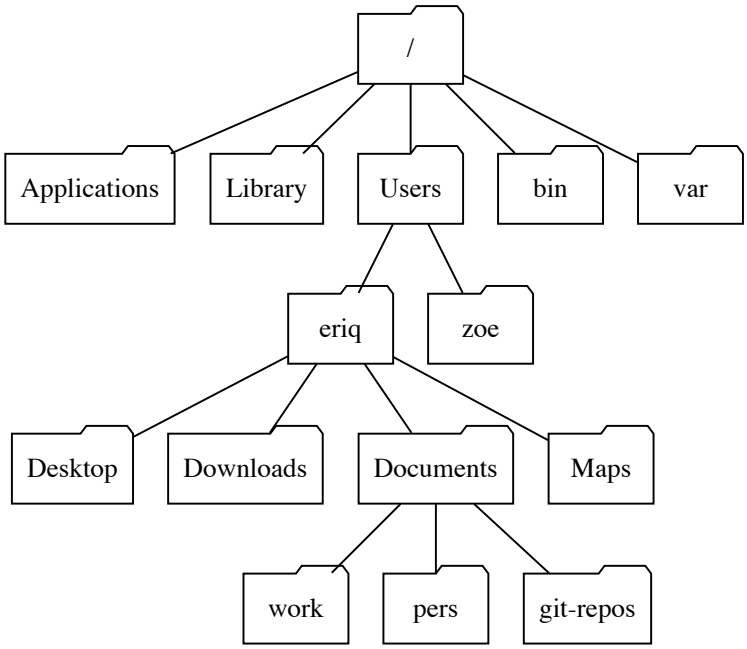


FIGURE 2.1: A partial view of the directories on the author's laptop.

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 8.2.) 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.

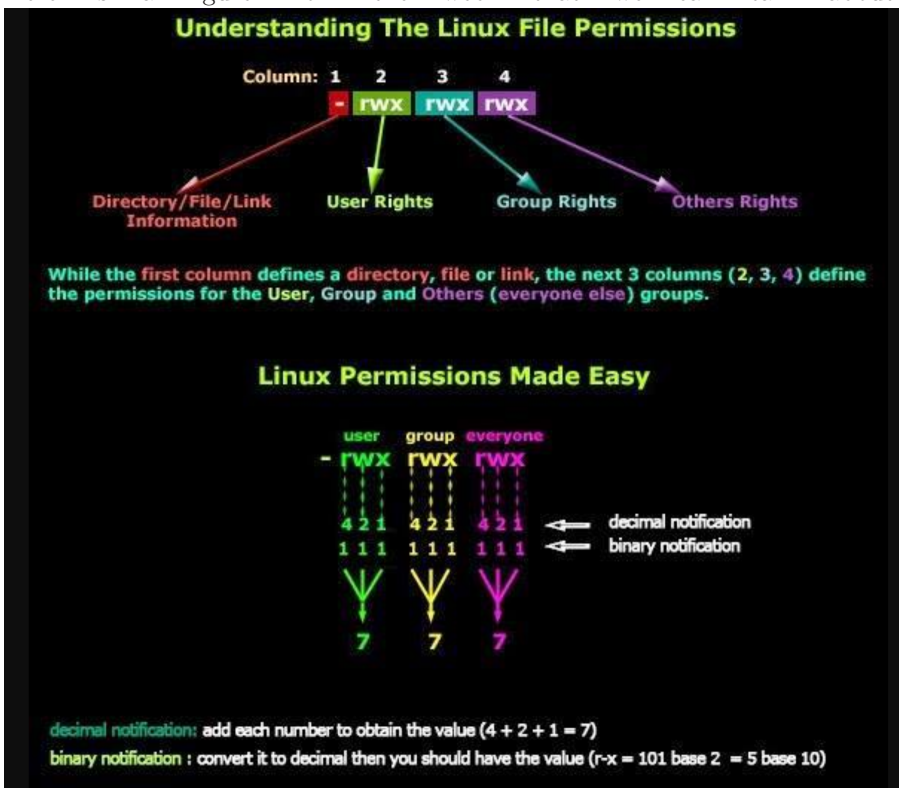
The Unix file system uses a system of permissions that gives rights to various classes of users to read, write, or execute files. The permissions associated with a file can be viewed using `ls -l`. They are captured in the first column which might look something like `-rwxr-xr-x`. When you first start 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

Discuss the programming interface, and also maybe discuss & and ; and how to put things into scripts.

In here, let's also talk about compression with gzip (and using `stuff | gzip -c > this.gz` to gzip and send to stdout.)

3.1 Advanced repitition

I want to get constructs like `{1..20}` and `{csv,pdf,jpg}` in here too.

3.2 Variables

3.3 looping

3.4 Further reading

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

3.5 reading files line by line

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

```
cat bwa-run-list.txt straggler-bwa-run-list.txt | while read -r line; do
  A=($line);
  file=${A[1]};
  num=${A[2]};
  du -h bam-slices/$file/${num}-sorted.bam;
done

2.3G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
2.2G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
2.2G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
2.3G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
2.2G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
2.2G    bam-slices/chinook_Battle.Creek.Sacramento.River_Schluter_GBC_001_CH1-2011_Male/00
```

Note that this is not how you want to rip through files, typically, because it is slow and awk is a much better bet. But, if you want to do a system call for each line, it ends up being a decent way forward.

3.6 Difference between double and single quotes

This becomes important when writing awk scripts and using bcftools expressions.

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; 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 tookbox.

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" operations in a text file.

5

High Performance Computing (HPC) Environments

Hey Eric! You might consider breaking this into two separate chatpers: 1 = working on remote computers and 2 = high-performance computing. The first could include all the stuff about scp, globus, rclone, and google drive.

This is going to be a chapter on using High Performance Computing clusters.

There is a lot to convey here about using queues and things.

I know SGE pretty well at this point, but others might use slurm.

Here is a good page with a comparison: https://confluence.csiro.au/display/SC/Reference+Guide%3A+Migrating+from+SGE+to+SLURM

And here is a good primer on SGE stuff: https://confluence.si.edu/display/HPC/Monitoring+your+Jobs

I guess I'll have to see what the CSU students have access to.

5.1 Accessing remote computers

Start off with some stuff about ssh and scp.

Maybe have a section on public and private keys so you don't have to put your password in every time (I would like to get better with that, as well).

Here is the deal on that. On a mac, you can use

```
ssh-keygen -t rsa -b 4096
```

and be sure to give it a password, so that your private key is not unencrypted. For, if it is, then there is a chance (I do believe) that if someone were to obtain that file, they could gain access to all the computers you are authorized on.

Note that a lot of tutorials on the web have you generating a private key without any encryption. That is lame.

Then copy the .ssh/id_rsa.pub key to the .ssh/authorized_keys file on the server (creating it if it needs to be there). Then ssh to the server and your Mac will pop up a window asking for a password or will prompt on the command line for one (note that the password stays local!). You put in the password that you used when you created the private key. That can be saved in the Mac keychain (on old version of OSX you might get asked if you want to save it in the Mac keychain). Voila! Now you have access to the server with no need to type a password in there.

Note, since Sierra, you need to add this to a `.ssh/config` file to get the password stored in the keychain:

```
Host *
   AddKeysToAgent yes
   UseKeychain yes
```

And, it turns out that you also need to make sure the permissions on that file are set appropriately:

```
chmod 600 ~/.ssh/config
```

It is actually pretty darn simple...

Note, to have access from another computer to the server, you probably just create a keypair for that computer, and add the public key to the authorized_keys.

Then add something like this to your .bashrc:

```
# for quick ssh to hoffman
alias hoffy='ssh eriq@hoffman2.idre.ucla.edu'

# to be used in scp.  like scp file $hoff:~/
hoff=eriq@hoffman2.idre.ucla.edu
```

Note that this should only be done on a private computer account, not on a shared account on a computer. Otherwise, everyone using that account will have access.

Also, an interesting thing to investigate might be SSHF/FUSE which just might let one mount the HPC filesystem on a local directory, so you can work with files there using RStudio, etc, get them all debugged, and then run them.

Check out some information about that here: https://www.digitalocean.com/community/tutorials/how-to-use-sshfs-to-mount-remote-file-systems-over-ssh.

I went to https://osxfuse.github.io/ and I downloaded SSHFS 2.5.0 and

FUSE for Mac OS X. Then I installed them, doing a simple default install for each.

Then, you make a mountpoint, which you can do in your home directory if you want:

```
mkdir ~/hoffman  # this is the mountpoint
sshfs -o allow_other,defer_permissions,IdentityFile=~/.ssh/id_rsa eriq@hoffman2.idre.ucla.

# close it out:
umount ~/hoffman/
```

Note that the absolute path to my home folder there seemed to be important.

Also, I can't get to `/u/nobackup/kruegg/eriq` from here by going through the symlink in my home directory on hoffman, since that is essentially a different volume. But I should be able to do this:

```
sshfs -o allow_other,defer_permissions,IdentityFile=~/.ssh/id_rsa eriq@hoffman2.idre.ucla.
```

Yep! that works. And I can open all those files from within RStudio. Cool.

Note, I will have to have a separate mount point for $SCRATCH as well.

Shoot! That works really seamlessly!

I can also add to my .bashrc:

```
alias hoffuse='sshfs -o allow_other,defer_permissions,IdentityFile=~/.ssh/id_rsa eriq@hoff
```

Now, the bad news: you can open an Rstudio project from that remote, mounted volume, but it doesn't seem to really work. It is incredibly slow, and then fails to quit properly, etc.

After force-quitting it, I am unable to unmount the hoffuse volume. What a mess. OK, I probably won't try (running the rstudio project over FUSE). But it might be useful still for editing small things.

5.2 Transferring files to remote computers

5.2.1 scp

5.2.2 Globus

5.2.3 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[1]. 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.

5.2.3.1 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`,

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

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 directory, 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.
- `-P`, `-v`, `-vv`: give me progress information, verbose output, or super-verbose output, respectively.
- `--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:
```

5.2.3.2 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 ./
```

5.2.3.3 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.

5.2.3.4 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.

5.2.3.5 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
```

5.2.3.6 Encrypting your config file

This makes sense, and it easy: with `rclone config` encryption is one of the options. When it is encrypted, use `rclone config show` to see what it looks like in clear text.

5.2.3.7 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.

5.2.4 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 5.2.3.2, 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 Activating/Installing software

5.3.1 Modules

This is if your sys admin has made it easy.

5.3.2 Miniconda

This is how one will probably want to do it

5.3.2.1 Testing this on Summit

I just want to quickly try this:

```
ssh eriq@colostate.edu@login.rc.colorado.edu

# get on compile nodes
ssh scompile

# I checked modules and found no samtools, bcftools, etc.

# Now install miniconda
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

# then you follow the prompts and agree to the license and the
# default install location.

## NOTE: Might want to install to a different location if there
## are serious caps on hard disk usage in home directory..

source ~/.bashrc

# after that I have it!
(base) [eriq@colostate.edu@shas0136 conda_install]$
```

Now, let's see if we can get samtools. Just google "samtools conda" to get an idea of how to do it:

```
conda install -c bioconda samtools
```

AFter that, it just works! Cool!

```
(base) [eriq@colostate.edu@shas0136 ~]$ samtools

Program: samtools (Tools for alignments in the SAM format)
Version: 1.9 (using htslib 1.9)

Usage:   samtools <command> [options]

Commands:
  -- Indexing
     dict           create a sequence dictionary file
     faidx          index/extract FASTA
     fqidx          index/extract FASTQ
     index          index alignment

...
```

All right! That is amazing. Next steps:

1. Tell students about establishing different environments.

2. Learn about how to make a minimal environment for a project and how to record that and be able to distribute/propagate it.

Along those lines, I want to see if, when I install samtools into a new environment, it re-downloads it or not…

```
conda create --name quick-test
conda activate quick-test

# i also made an environment in a specified directory (you
# could put these within a project directory)
conda create --prefix ./test-envs-dir
conda activate ./test-envs-dir

# now, let's install bcftools there
conda install -c bioconda bcftools

# note that it doesn't re-download the dependencies, as far as I can tell.
conda activate base
bcftools # not found in environment base

conda activate ./test-envs-dir
bcftools  # it is found in this environment.  Cool.

conda activate quick-test
bcftools  # it ain't here
```

Now, after that, bcftools is in ./test-envs-dir/bcftools

So, what if we install it into another environment. Does it symlink it?

```
conda activate base
conda install -c bioconda bcftools
bcftools

# whoa! That errored out!
bcftools: error while loading shared libraries: libcrypto.so.1.0.0: cannot open shared obj

# that is a serious problem.

# Can I get it in my other environment?
conda activate quick-test
conda install -c bioconda bcftools
bcftools
```

```
# that totally works, but it still doesn't in base...

# so, what if we add samtools to our other environments?
# that works fine.
```

But, samtools/bcftools dependency issues are a known problem: https://github.com/sunbeam-labs/sunbeam/issues/181 Basically they rely on different versions of some ssh libs.

Note that installing bcftools first things work. But what if we make another environment and install samtools first again?

```
conda create --name samtools-first
conda activate samtools-first
conda install -c bioconda samtools  # this didn't download anything new
conda install -c bioconda bcftools

# BOOM! THIS CREATES A FAIL.  SO, YOU GOTTA INSTALL BCFTOOLS
# FIRST.  FAR OUT.
```

5.3.3 Exporting environments

Looks like we should be able to do this. Let's do the one that works:

```
(quick-test) [~]--% conda env export > quick-test-env.yml

(quick-test) [~]--% cat quick-test-env.yml
name: quick-test
channels:
  - bioconda
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - bcftools=1.9=ha228f0b_4
  - bzip2=1.0.8=h7b6447c_0
  - ca-certificates=2019.5.15=1
  - curl=7.65.3=hbc83047_0
  - htslib=1.9=ha228f0b_7
  - krb5=1.16.1=h173b8e3_7
  - libcurl=7.65.3=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=1.1.1c=h7b6447c_1
  - samtools=1.9=h10a08f8_12
  - tk=8.6.8=hbc83047_0
  - xz=5.2.4=h14c3975_4
  - zlib=1.2.11=h7b6447c_3
prefix: /home/eriq@colostate.edu/miniconda3/envs/quick-test


# OK, that is cool.  Now, if we wanted to email that to someone,
# we could, and then they could do this:
conda env create --name dupie-quick -f quick-test-env.yml

# that environment then has samtools and bcftools

# note that it probably would try to name it quick-test if we didn't
# pass in the name there...
```

5.4 Boneyard

This is a variant of rsync that let's you sync stuff up to google drive. It might be a better solution than rcp for getting stuff onto and off of google drive. Here is a link: https://rclone.org/. I need to evaluate it. It might also be a good way to backup some of my workstuff on my laptop to Google Drive (and maybe also for other people to create replicas and have a decent backup if they have unlimited Google Drive storage).

I got this working. It is important to set your own OAuth client ID: https://forum.rclone.org/t/very-slow-sync-to-google-drive/6903

After that I did like this:

```
rclone sync -vv --tpslimit 10 --fast-list Otsh_v1.0_genomic.fna  gdrive-rclone:spoogee-spo
```

which did 2 Gb of fasta into the spoogee-spoogee directory pretty quickly.

But, with something that has lots of files, it took longer:

```
# this is only about 100 Mb but took a long time
rclone copy -P --tpslimit 10 --fast-list  rubias  gdrive-rclone:rubias
```

However, once that is done, you can sync it and it finds that parts that have changed pretty quickly.

it appears to do that by file modification times:

```
2019-04-19 23:21 /Otsh_v1.0/--% rclone sync -vv --tpslimit 10 --fast-list Otsh_v1.0_genomi
2019/04/19 23:21:36 DEBUG : rclone: Version "v1.47.0" starting with parameters ["rclone" "
2019/04/19 23:21:36 DEBUG : Using config file from "/Users/eriq/.config/rclone/rclone.conf
2019/04/19 23:21:36 INFO  : Starting HTTP transaction limiter: max 10 transactions/s with
2019/04/19 23:21:37 DEBUG : GCF_002872995.1_Otsh_v1.0_genomic.gff.gz: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.dict: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.amb: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.ann: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.bwt: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.fai: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.pac: Excluded
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna.sa: Excluded
2019/04/19 23:21:37 INFO  : Google drive root 'spoogee-spoogee': Waiting for checks to fin
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna: Size and modification time the same (di
2019/04/19 23:21:37 DEBUG : Otsh_v1.0_genomic.fna: Unchanged skipping
2019/04/19 23:21:37 INFO  : Google drive root 'spoogee-spoogee': Waiting for transfers to
2019/04/19 23:21:37 INFO  : Waiting for deletions to finish
2019/04/19 23:21:37 INFO  :
Transferred:              0 / 0 Bytes, -, 0 Bytes/s, ETA -
Errors:                 0
Checks:                 1 / 1, 100%
Transferred:            0 / 0, -
Elapsed time:         1.3s

2019/04/19 23:21:37 DEBUG : 5 go routines active
2019/04/19 23:21:37 DEBUG : rclone: Version "v1.47.0" finishing with parameters ["rclone"
2
```

So, for moving big files around that might be a good way forward. I will have to do a test with some big files.

And I need to test it with team drives so that multiple individuals can pull stuff off of the Bird Genoscape drive for example.

It would be nice to have safeguards so people don't trash stuff accidentally....

5.4.0.1 rclone on Hoffman

Their default install script expects sudo access to put it in /usr/local but I don't on hoffman, obviously, so I just downloaded the the install script and edited the section for Linux to look like this at the relevant part

```
case $OS in
  'linux')
    #binary
    cp rclone ~/bin/rclone.new
    chmod 755 ~/bin/rclone.new
    #chown root:root /usr/bin/rclone.new
    mv ~/bin/rclone.new ~/bin/rclone
    #manuals
    #mkdir -p /usr/local/share/man/man1
    #cp rclone.1 /usr/local/share/man/man1/
    #mandb
    ;;
```

I don't get man pages, but I get it in ~/bin no problem.

To set up the configuration, check where it belongs:

```
% rclone config file
Configuration file doesn't exist, but rclone will use this path:
/u/home/e/eriq/.config/rclone/rclone.conf
```

And then I just put my config file from my laptop on there. I just pasted the stuff in whilst emacsing it. Holy cow! That is super easy.

Note that the config file is where you can also set default options like tpslimit and fast-list I think.

So, the OAuth stuff is all stored in that config file. And if you can set it up on one machine you can go put it on any others that you want. That is awesome.

When it was done, I tested it:

```
% rclone sync -vv  --drive-shared-with-me  gdrive-rclone:BaselinePaper  BaselinePaper_here
2019/04/29 14:49:24 DEBUG : rclone: Version "v1.47.0" starting with parameters ["rclone" "
2019/04/29 14:49:24 DEBUG : Using config file from "/u/home/e/eriq/.config/rclone/rclone.c
2019/04/29 14:49:25 INFO  : Local file system at /u/home/e/eriq/BaselinePaper_here: Waitin
2019/04/29 14:49:25 INFO  : Local file system at /u/home/e/eriq/BaselinePaper_here: Waitin
2019/04/29 14:49:26 DEBUG : Local file system at /u/home/e/eriq/BaselinePaper_here: File t
2019/04/29 14:49:26 DEBUG : BaselinePaper_Body.docx: Failed to pre-allocate: operation not
```

```
2019/04/29 14:49:26 INFO  : BaselinePaper_Body.docx: Copied (new)
2019/04/29 14:49:26 DEBUG : BaselinePaper_Body.docx: Updating size of doc after download t
2019/04/29 14:49:26 INFO  : BaselinePaper_Body.docx: Copied (Rcat, new)
2019/04/29 14:49:27 DEBUG : Local file system at /u/home/e/eriq/BaselinePaper_here: File t
2019/04/29 14:49:27 DEBUG : ResponseToReviewers_eca.docx: Failed to pre-allocate: operatio
2019/04/29 14:49:27 INFO  : ResponseToReviewers_eca.docx: Copied (new)
2019/04/29 14:49:27 DEBUG : ResponseToReviewers_eca.docx: Updating size of doc after downl
2019/04/29 14:49:27 INFO  : ResponseToReviewers_eca.docx: Copied (Rcat, new)
2019/04/29 14:49:27 INFO  : Waiting for deletions to finish
2019/04/29 14:49:27 INFO  :
Transferred:      193.543k / 193.543 kBytes, 100%, 79.377 kBytes/s, ETA 0s
Errors:                 0
Checks:                 0 / 0, -
Transferred:            4 / 4, 100%
Elapsed time:        2.4s

2019/04/29 14:49:27 DEBUG : 6 go routines active
2019/04/29 14:49:27 DEBUG : rclone: Version "v1.47.0" finishing with parameters ["rclone"
```

That was fast and super solid.

5.4.0.2 Encrypt the config file

You can use `rclone config edit` to set a password for the config file. Then it encrypts that so no one is able to run wild if they just get that file. You have to provide your password to do any of the rclone commands. If you want to see the config file use `rclone config show`. You could always copy that elsewhere, and then re-encrypt it.

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.

5.5 The Queue (SLURM/SGE/UGE)

5.6 Modules package

5.7 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.

5.8 Job arrays

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.

5.9 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…)

5.10 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

6

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)

7

Rstudio and Project-centered Organization

Somewhere talk about here::here(): https://github.com/jennybc/here_here

7.1 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.

8

Version control

8.1 Why use version control?

8.2 How git works

8.3 git workflow patterns

8.4 using git with Rstudio

8.5 git on the command line

9

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

10

Authoring reproducibly with Rmarkdown

10.1 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.

10.2 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

10.2.1 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

10.3 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
```

10.4 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.

11

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

12

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.

13

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.

13.1 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 13.1 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:

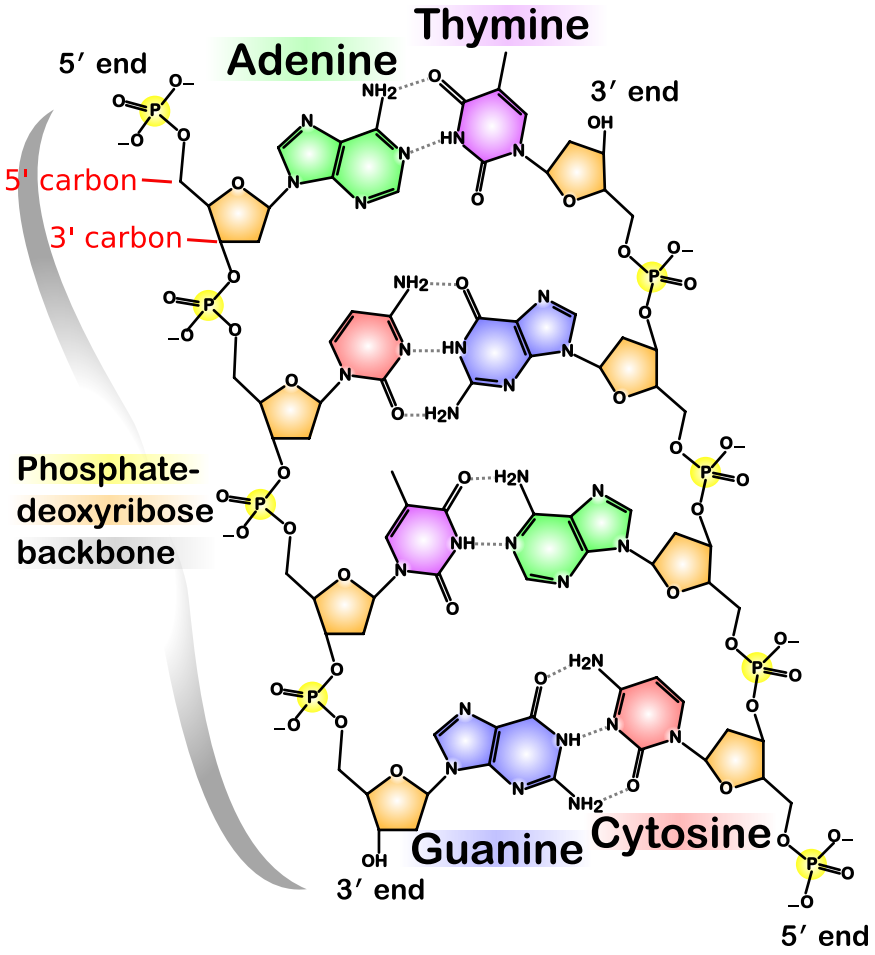


FIGURE 13.1: Schematic of the structure of DNA. (Figure By Madprime (talk—contribs) CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1848174)

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 13.1.

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'
```

13.1.1 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 13.2).

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 13.2: 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 13.3 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 13.3, 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

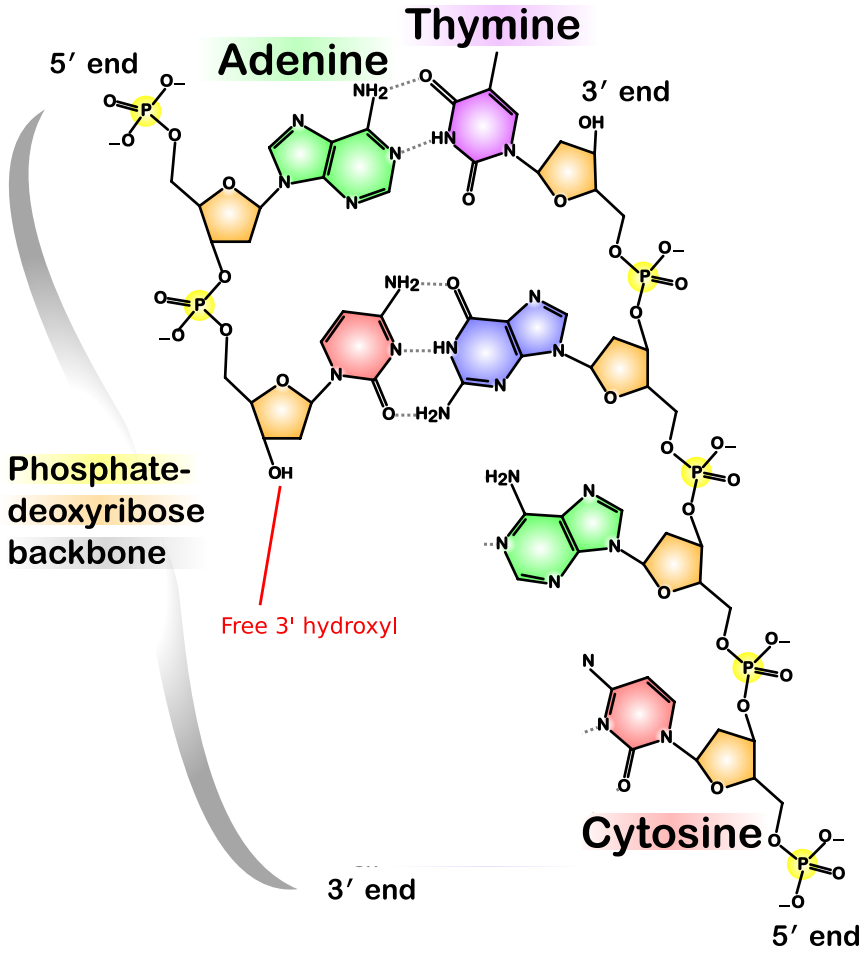


FIGURE 13.3: DNA during replication. (Figure adapted from the one by Madprime (talk—contribs) CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1848174)

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.

13.1.2 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.

13.2 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:

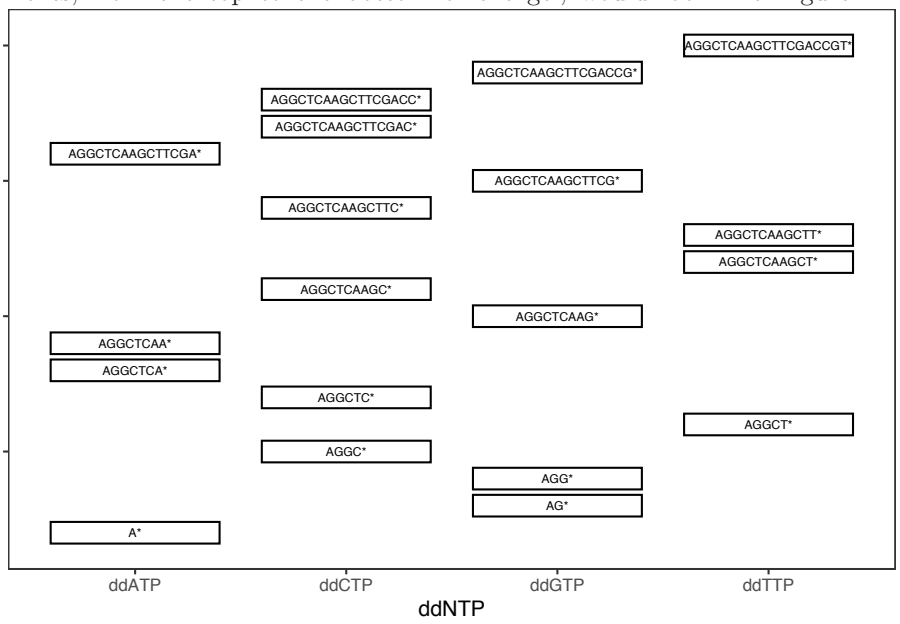
```
## [1] "AG*"                 "AGG*"
## [3] "AGGCTCAAG*"          "AGGCTCAAGCTTCG*"
## [5] "AGGCTCAAGCTTCGACCG*"
```

And in a final, separate reaction, incorporation of ddTTP would give:

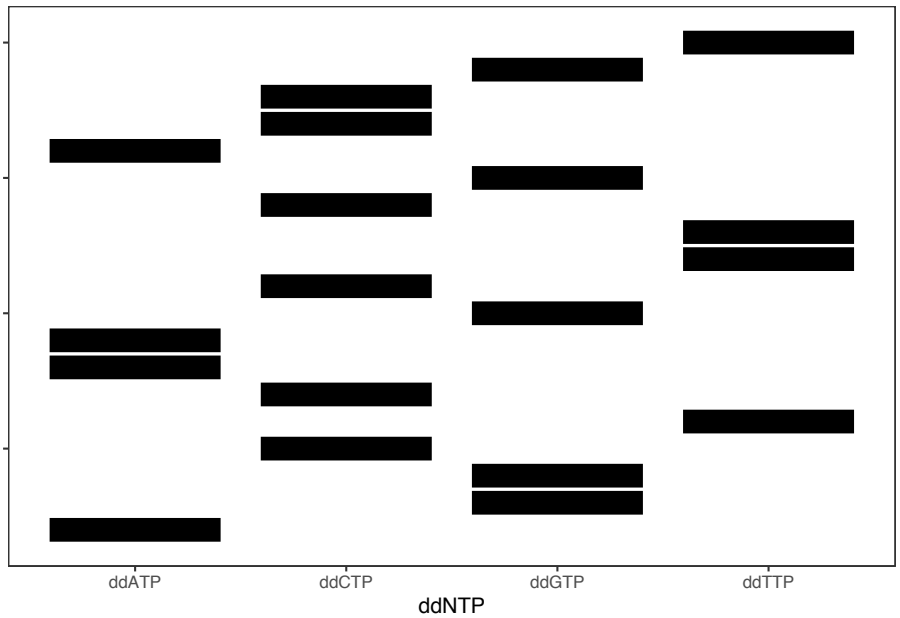
```
## [1] "AGGCT*"               "AGGCTCAAGCT*"
```

```
## [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.

13.3 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.

13.4 Library Prep Protocols

Gotta mention here about how barcodes work.

How you prepare your libraries dictates what type of data you get.

13.4.1 WGS

13.4.2 RAD-Seq methods

13.4.3 Amplicon Sequencing

13.4.4 Capture arrays, RAPTURE, etc.

14

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.

14.1 Sequences

14.2 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.

14.2.1 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?

14.2.2 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 14.1.

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?

14.2.3 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.

14.2.3.1 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 14.1: 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.

14.2.3.2 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
```

14.2.3.3 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…

14.2.3.4 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)
```

14.2.3.5 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.

14.2.4 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…

14.3 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.

14.3.1 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.

14.3.2 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
```

14.4 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.

14.4.1 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'
    ||||||||||||||||||||||||||||||||||||||||||||||||||
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'
```

[1] https://samtools.github.io/hts-specs/SAMv1.pdf

```
                ||||||||||||||||||||||||||||||||||||||||||||||||
            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 14.1 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 14.1: 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

Column	Field	Data Type	Description
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

14.4.2 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.

14.4.3 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 14.2, which also lists the meaning of each bit.

TABLE 14.2: 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
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 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

[2] https://broadinstitute.github.io/picard/explain-flags.html

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.

14.4.4 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 3' 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 be2M2D4M4I2M‘ 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).

14.4.5 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 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.

14.4.6 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

[3]https://cran.r-project.org/web/packages/microhaplot/index.html

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.

This section not yet complete.

14.4.7 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 standard Unix tools. If you want to view a BAM file, you can use `samtools view` to read it in SAM format.

14.4.8 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'
```

14.5 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.

14.6 Segments

BED

14.7 Conversion/Extractions between different formats

- vcflib's vcf2fasta takes a phased VCF file and a fasta file and spits out sequence.

14.8 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.

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").

14.8.1 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?

15

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.

16

Alignment of sequence data to a reference genome

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.

16.1 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"

16.2 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.

16.3 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.

16.4 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.

17

Variant calling with GATK

Standard stuff here.

Big focus on parallelizing.

18

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.

19

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.

20

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.

21

Whole genome alignment strategies

Basically want to talk about situations

21.1 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.)

21.2 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....)

21.2.1 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?

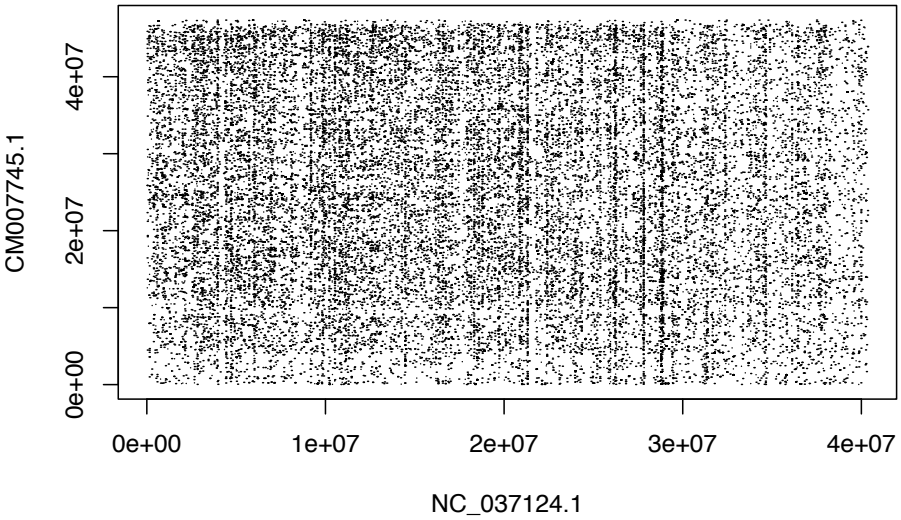
```
time lastz chinook-28.fna coho-28.fna --notransition --step=20 --nogapped --ambiguous=iupa
```

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.

21.2.2 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!

21.2.3 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
```

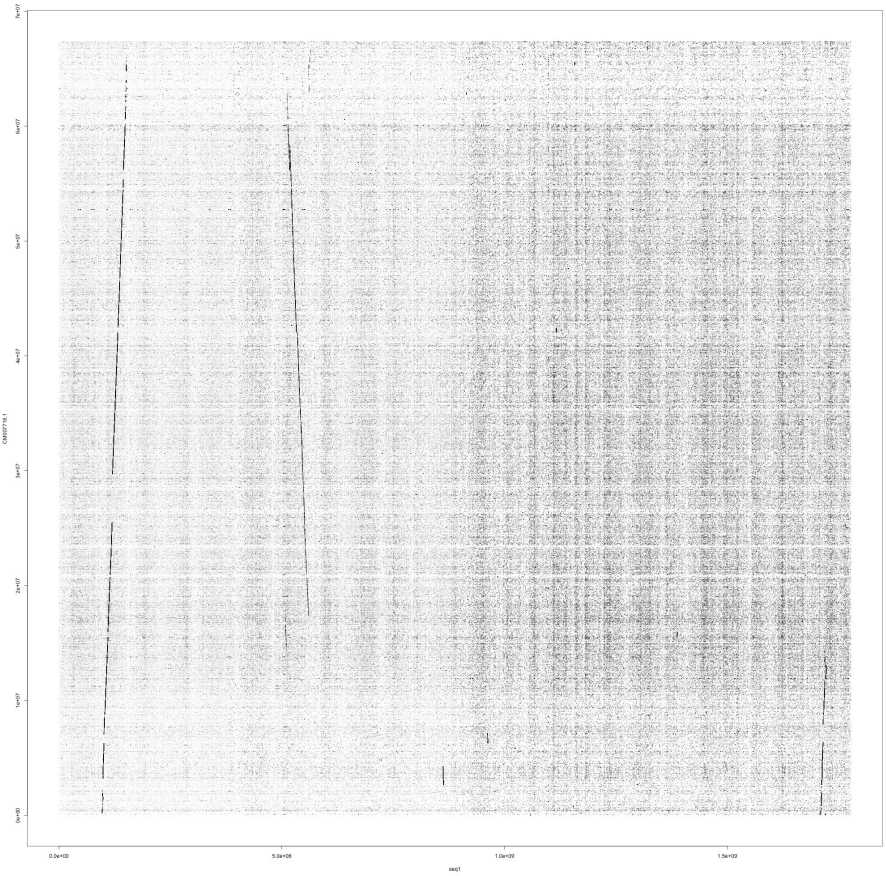


FIGURE 21.1: Coho Chromo 1 on catenated chinook chromos

```
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")
```

21.2.3.1 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.

21.2.3.2 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.

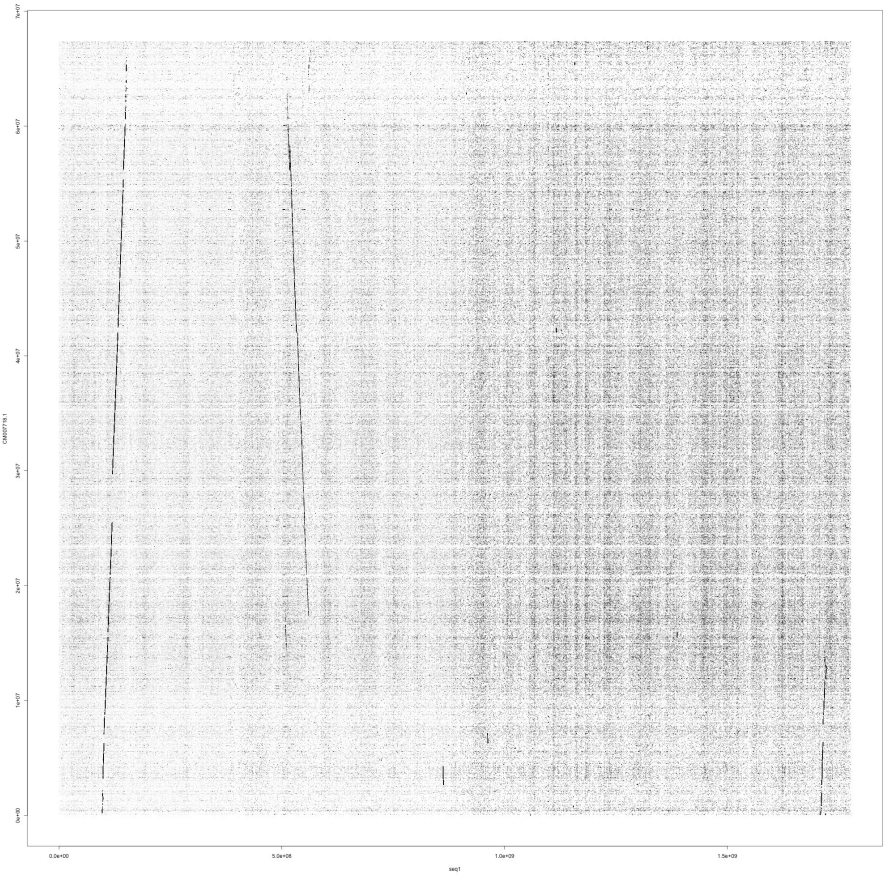


FIGURE 21.2: Coho Chromo 1 on catenated chinook chromos. Ident=99.5

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

22

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

23

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.

23.1 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.

23.2 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.

23.3 Demographic inference with $\partial a \partial i$ and *moments*

23.4 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)

23.5 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.

23.6 Relationship Inference

Maybe do a lecture on this…

23.7 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.

23.8 Multivariate Associations, GEA, etc.

It really is time for me to wrap my head around this stuff.

23.9 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.