Eric C. Anderson

Practical Computing and Bioinformatics for Conservation and Evolutionary Genomics

To my son,

without whom I should have finished this book two years earlier

Contents

List of Tables

List of Figures

Preface

Hi there, this is my great book.

Why read this book

It is very important...

Structure of the book

Chapters **??** introduces a new topic, and ...

Software information and conventions

I used the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2018) to compile my book. My R session information is shown below:

```
xfun::session_info()
```

```
## R version 3.6.0 (2019-04-26)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Sierra 10.12.6
##
## Locale: en_US.UTF-8 / en_US.UTF-8 / en_US.UTF-8 / C / en_US.UTF-8 / en_US.UTF-8
##
## Package version:
##   askpass_1.1        assertthat_0.2.1
##   backports_1.1.4    base64enc_0.1.3
##   BH_1.69.0.1        bookdown_0.10
##  broom_0.5.2        callr_3.2.0
##  cellranger_1.1.0   cli_1.1.0
##  clipr_0.6.0        colorspace_1.4-1
##  compiler_3.6.0     crayon_1.3.4
##  curl_3.3           DBI_1.0.0
##  dbplyr_1.4.0       digest_0.6.19
##  dplyr_0.8.1        ellipsis_0.1.0
##  evaluate_0.13      fansi_0.4.0
##  forcats_0.4.0      fs_1.3.1
##  generics_0.0.2     ggplot2_3.1.1
##  glue_1.3.1         graphics_3.6.0
##  grDevices_3.6.0    grid_3.6.0
##  gtable_0.3.0       haven_2.1.0
##  highr_0.8          hms_0.4.2
##  htmltools_0.3.6    httr_1.4.0
##  jsonlite_1.6       knitr_1.23
##  labeling_0.3       lattice_0.20-38
##  lazyeval_0.2.2     lubridate_1.7.4
##  magrittr_1.5       markdown_0.9
##  MASS_7.3.51.4      Matrix_1.2.17
##  methods_3.6.0      mgcv_1.8.28
##  mime_0.6           modelr_0.1.4
##  munsell_0.5.0      nlme_3.1-139
##  openssl_1.3        pillar_1.4.0
##  pkgconfig_2.0.2    plogr_0.2.0
##  plyr_1.8.4         prettyunits_1.0.2
##  processx_3.3.1     progress_1.2.2
##  ps_1.3.0           purrr_0.3.2
##  R6_2.4.0           RColorBrewer_1.1.2
##  Rcpp_1.0.1         readr_1.3.1
##  readxl_1.3.1       rematch_1.0.1
##  reprex_0.3.0       reshape2_1.4.3
##  rlang_0.3.4        rmarkdown_1.13
##  rstudioapi_0.10    rvest_0.3.4
##  scales_1.0.0       selectr_0.4.1
##  splines_3.6.0      stats_3.6.0
##  stringi_1.4.3      stringr_1.4.0
##  sys_3.2            tibble_2.1.1
##  tidyr_0.8.3        tidyselect_0.2.5
##  tidyverse_1.2.1    tinytex_0.13
##  tools_3.6.0        utf8_1.1.4
##  utils_3.6.0        vctrs_0.1.0
##  viridisLite_0.3.0  whisker_0.3.2
##  withr_2.1.2        xfun_0.7
##  xml2_1.2.0         yaml_2.2.0
##    zeallot_0.1.0
```

Package names are in bold text (e.g., **rmarkdown**), and inline code and filenames are formatted in a typewriter font (e.g., `knitr::knit('foo.Rmd')`). Function names are followed by parentheses (e.g., `bookdown::render_book()`).

Acknowledgments

A lot of people helped me when I was writing the book.

Frida Gomam
on the Mars

0

Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter . If you do not manually label them, there will be automatic labels anyway, e.g., Chapter **??**.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```r
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

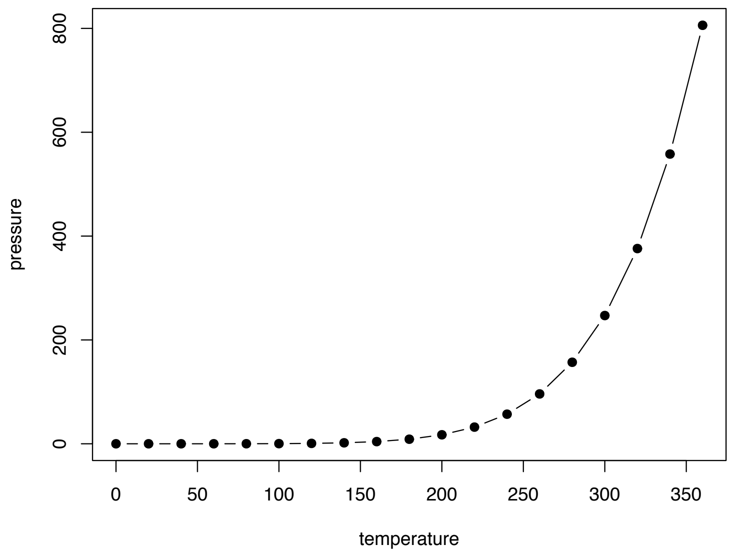


FIGURE 1: Here is a nice figure!

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 0.1.

TABLE 0.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2018) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

And John wrote a paper: (Novembre and Barton, 2018)

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?

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 bioinformatc 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!

A paragraph about what kinds of computers run Unix. (Often big servers, but also some flavors of personal laptops). The difference (similarities) between Unix and Linux. You might have Unix on your system (if you have a Mac). Or not. All this will effect what you need to do to get a Unix shell on your computer

2.1 Getting a bash shell on your system

Talk about what a shell is.

For windows, it looks like windows 10 might be able to do pretty well with it: https://www.howtogeek.com/265900/everything-you-can-do-with-windows-10s-new-bash-shell/

But maybe I should get a windows box and use Rstudio and see how the terminal in R studio is.

For now, I will punt on this and get just assume POSIX available on the system.

However, we will want to start everyone off by having them type `bash` to make sure they have a bash shell.

Also, I am going to have to talk a little bit about what a shell is here….

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.

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

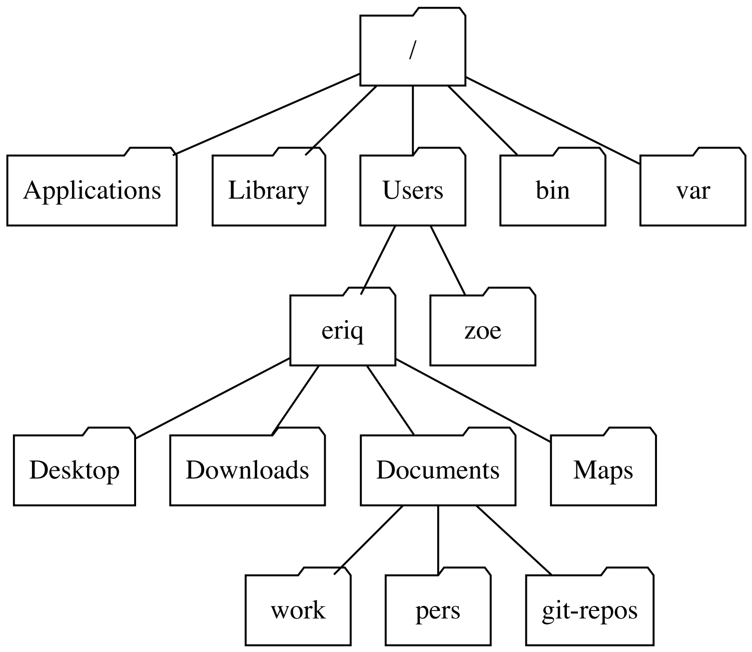


FIGURE 2.1: A partial view of the directories on the author's laptop.

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

Gotta talk about the `-d` option.

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, `?`.

Should make it clear that this is not something unique only to ls. All that ls is doing is listing its arguments. The expansion happens on the command line.

Gonna have to talk about `-l` and `-a` options. Maybe talk about -d, which is useful.

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!

So, to make your life easier, 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

In general things look like this:

`command` *options arguments*

Talk about –help and man pages.

Discuss symbolic links — a special kind of file. Maybe do this when discussing file types.

Then introduce `open` and a PC bash equivalent if available.

2.3.1 Exercises

Exercise 2.1 (Some exercise)**.** We can't make a fenced block. But we coul say, Consider the following

```
ls all*.txt ball*.txt ala*.txt
```

And then see how that comes out.

Exercise 2.2 (Some other exercise)**.** Here is another.

2.4 Handling, Manipulating, and Viewing files and streams

2.4.1 Fundamental file-handling commands

`cp`, `mv`, 'rm'

Note the difference between cp and mv and how they behave with directories too. Maybe talk a little about links.

2.4.2 Everytyhing is a file. I/O: redirection and pipes

Three major types of files: regular files, directories, and symbolic links.

Executable files.

Somewhere, at some point I will have to talk about chmod and about what the hell 600 is, for example (u+rw). So, a whole little session on permissions, and I should also explain the nomenclature like 755. An exercise where students touch a file in /tmp and then use the numbers of u+x type commands to get an end result (like `-rw- --- ---`). For example: *use chmod to get a result like this*: `-rwxrw-r--`.

`chmod 764`

Note that the first position denotes whether it is a directory or not.

Everything is a file, some of them are executuble (programs) and they typically do things to other files.

Unix philosophy of spitting text out so that stuff can be done to it (chaining/piping) or redirecting.

`stdin`, `stdout`, `stderr`.

2.5 Customizing your Environment

Gotta start off by talking about PATHs and things when you talk about. Then maybe some good tips for customization.

2.6 A short list of useful commands.

cat, head (good for binary files too with -c), less, sort, paste, cut, tar, gzip, du, wc, date

2.7 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

```
## Warning: `as_tibble.matrix()` requires a matrix with column names or a `.name_repair` a
## This warning is displayed once per session.
```

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

absolute path	relative path	`/` at beginning of path
`/` between directories	home directory	`~`
current working directory	**pwd**	`cd`
`.`	`..`	`cd -`

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

gotta talk about these bad boys...

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 anyone 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`, 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

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

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 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.

5.2.3.3 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.

5.2.3.4 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 with rclone config.

5.2.3.5 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.

5.3 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….

5.3.0.1 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.

5.3.0.2 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.

5.4 The Queue (SLURM/SGE/UGE)

5.5 Modules package

5.6 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.

5.7 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.

5.8 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…)

5.9 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 depen-*

dencies. 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

10.2.1 citr and Zotero

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)

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:

[1] `https://forums.zotero.org/discussion/61715/prevent-extra-braces-in-bibtex-export`

```
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

Next Generation Sequencing Data Acquisition

13.1 DNA Stuff

We are primarily going to be interested in representing DNA molecules in text format as series of nucleotide bases (A, C, G, T). But, in order to do this in a consistent fashion that lets us correctly and precisely compare one sequence of DNA to another we need to know how DNA "works" and we require some conventions.

Talk about the fact that it is double-stranded, and hence there is a reverse complement.

Genome references and forward vs reverse strand stuff.

13.2 Sequencing platforms

Basically talk a little about how the technologies work.

Go into paired-end vs single end reads, etc.

Strand direction. There is a good overview on a blog at: `http://www.cureffi.org/2012/12/19/forward-and-reverse-reads-in-paired-end-sequencing/`, to start getting one's head around it. It ends up being super important if you are looking for inversion breakpoints, etc (Corbett-Detig et al., 2012).

13.3 Library Prep Protocols

Gotta mention here about how barcodes work.

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

13.3.1 WGS

13.3.2 RAD-Seq methods

13.3.3 Amplicon Sequencing

13.3.4 Capture arrays, RAPTURE, etc.

14

Bioinformatic file formats and associated tools

Go over these, and for each, pay special attention to compressed and indexed forms and explain why that is so important. And I think that I should probably talk about the programs that are available for manipulating each of these.

14.1 Sequences

FASTA, FASTQ

14.2 Alignments

BAM. Talk about them being sorted or not. Note that the headers between these and VCF are similar.

Have a discussion here about naming positions within the genome and talk about 0-based vs 1-based coordinates.

14.3 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.

14.4 Segments

BED

14.5 Conversion/Extractions between different formats

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

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 no 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.

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_0C
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?

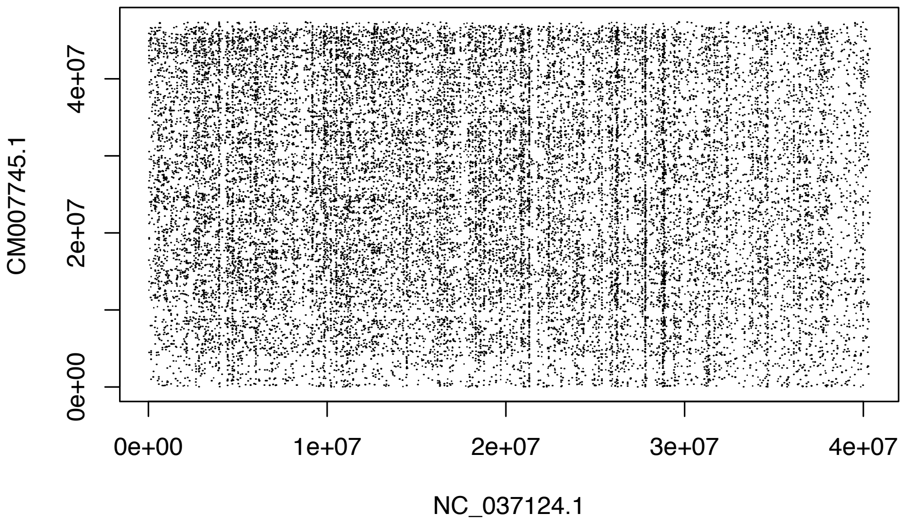
```
time lastz chinook-28.fna coho-28.fna --notransition --step=20 --nogapped --ambiguous=iupa
```

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
real    0m29.055s
```

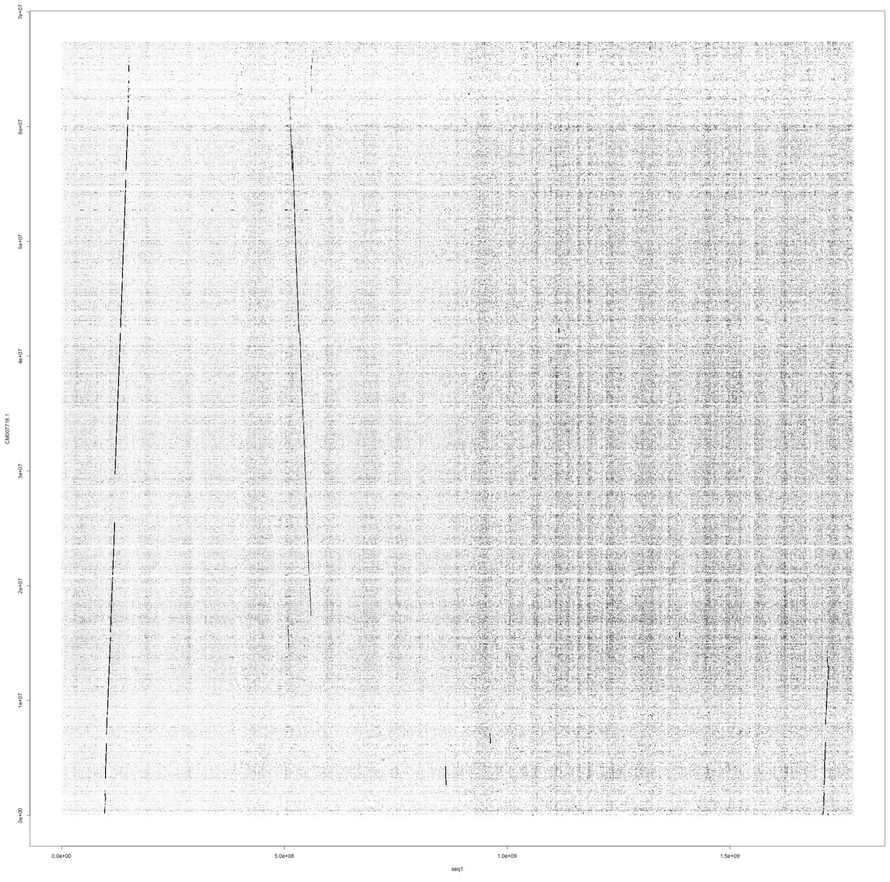


FIGURE 21.1: Coho Chromo 1 on catenated chinook chromos

```
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.
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

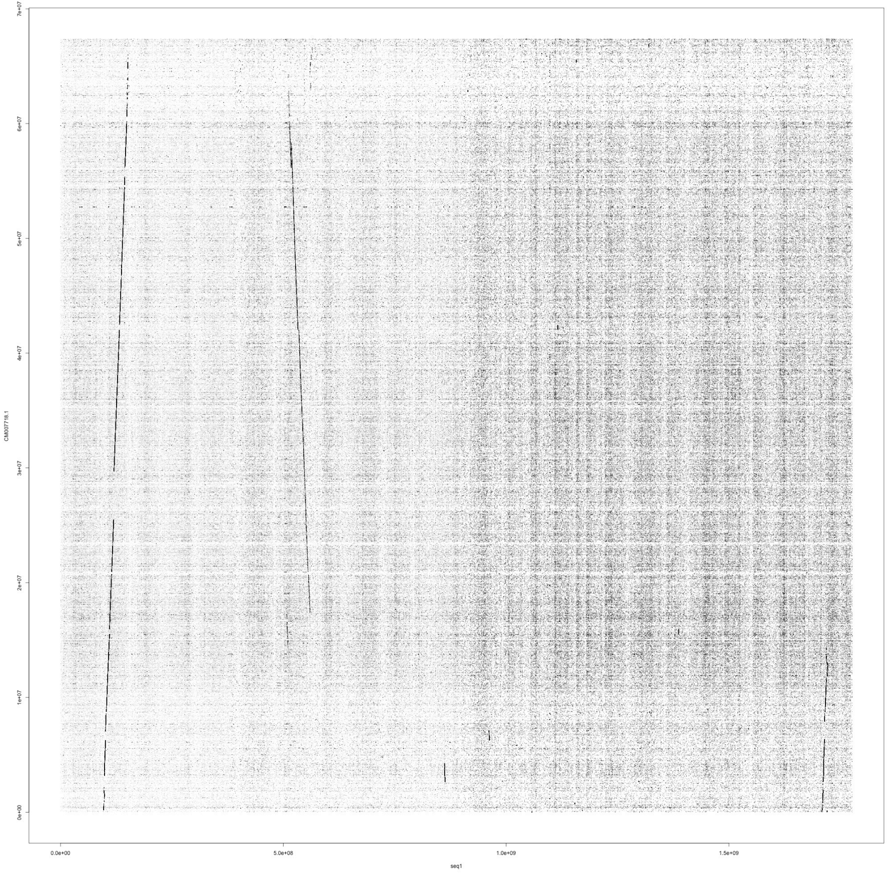


FIGURE 21.2: Coho Chromo 1 on catenated chinook chromos. Ident=99.5

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 d_{xy}, F_{ST}.

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.

Corbett-Detig, R. B., Cardeno, C., and Langley, C. H. (2012). Sequence-Based Detection and Breakpoint Assembly of Polymorphic Inversions. *Genetics*, 192(1):131–137.

Novembre, J. and Barton, N. H. (2018). Tread Lightly Interpreting Polygenic Tests of Selection. *Genetics*, 208(4):1351–1355.

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.

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.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.9.1.

Index