Introduction to Bioinformatics - 5023Y

Philip Leftwich

2021-04-26

Contents

Chapter 1

Introduction

This is a very short introduction to some of the processes and tools we use when working in bioinformatics.

This is not meant to be anywhere near comprehensive, you will be shown a few of the fundamentals, and one walkthrough of a single bioinformatics 'pipeline'.

The *main* difference between Bioinformatics and the data analysis you have done before is the **size** of the data. Bioinformatics usually deals with sequencing data, and this data has large file sizes.

Handling **big data** means you need to know how to operate on a supercomputer so we will be learning a little bit of Linux, processing **big** data into **small** data and then exporting into R to make some more amazing data visuals and interpret our findings!

DON'T worry if you don't understand everything

DO ask lots of questions!

Chapter 2

Unix

Unix is very likely the most fundamental skillset we can develop for bioinformatics (and much more than bioinformatics). Many of the most common and powerful bioinformatics approaches happen in this text-based environment, and having a solid foundation here can make everything we're trying to learn and do much easier. This is a short introductory tutorial to help us get from being completely new to Unix up to being friendly with it

2.1 What is Unix/Linux?

UNIX is a computer operating system. It was first developed in 1969 at Bell Labs. Unix is written in the programming language `C`.

Unix is proprietary software, whereas Linux is *basically* free and open-source Unix.

The Linux Operating System is highly flexible, free, open-source (like R) and uses very little RAM to run (Unlike Windows OS) - as such you find most supercomputers run on Linux. Operationally Linux is almost identical to Unix, and so we often refer to it under the umbrella term of "unix-Like" systems.

2.1.1 Some terms

Here are some terms worth knowing, don't worry about memorising them, it can just be useful to have these to refer to in the future.

Term	What it is
shell	what we use to talk to the computer; anything where you are pointing and clicking with a mouse
command line	a text-based environment capable of taking input and providing output
Terminal	A program that runs a shell

Unix	a family of operating systems (we also use the term “Unix-like” because one of the m
Linux	a ”Unix-like” OS
bash	the most common programming language used at a Unix command-line
flag	a way to set options for a function, a specific type of argument usually preceded by a

Note

You should be very familiar with using a GUI (RStudio), but remember we have spent a lot of time working with files and directories using the the command line (CLI) in R. This is useful practice, because most supercomputers lack a GUI, you must work entirely using the command line.

2.2 Why Learn Unix?

Most sequencing data files are large, and require a lot of RAM to process. As a result most of the work Bioinformaticians do is not hosted on their own computers, instead they “remote-connect” to high performance supercomputers or cluster computers. Almost all of these high performance computers use “Unix-like” operatings systems, the most common of which is Linux.

As stated above Linux is free (so no expensive licenses), open-source so lots of developers, its also well known for being stable, secure, reliable and efficient.

You already have some experience with using a Linux OS - every time you log into RStudio Cloud you are connecting to a supercomputer that runs on Linux. Normally we do not interact directly with the OS, instead we use R and RStudio directly.

But when you click on the RStudio Terminal it provides direct access to a command-line where we can execute commands and functions directly in Linux.

This allows us to start using programs other than R, and potentially use multiple programs & programming languages to work together.

Note

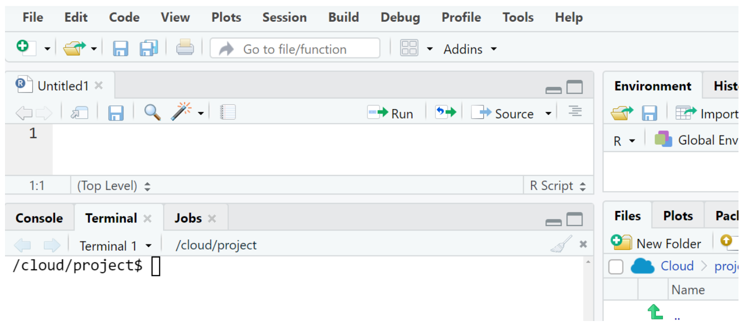
This series of practicals is designed for you to have a first introduction to Bioinformatics, it’s about exposure, not memorising or mastering anything. Don’t worry about the details!

2.3 Getting started

Before we get started we need a terminal to work in.

- Open the Bioinformatics RStudio Cloud Project in the 5023Y workspace

- Click on the `Terminal` tab next to `Console` in the bottom-left pane of the RStudio GUI, this opens a command-line *Shell*



```
/cloud/project$
```

Figure 2.1: Here is an example of the Terminal tab, right next to the R console

- This is our "command line" where we will be typing all of our commands. We type our commands in `bash`
- The $ is where you start typing from, left of this it tells you what folder you are currently in (`working directory`)
- If you need to, you can exit the Terminal and start a new session easily with options in RStudio

2.4 A few foundational rules

- Just like in R spaces are special, spaces break things apart, as a rule it is therefore better to have functions and file names with dashes (-) or underscores (_) - e.g. "draft_v3.txt" is preferred to "draft v3.txt".
- The general syntax on the command line is: `command argument`. Again this is very similar to R except we don't use brackets e.g. in R we are used to `command(argument)`
- Arguments can be **optional** e.g. if their is a default argument you may not have to write anything. Some functions *require* that arguments are specified. Again this is just like R.

2.5 Let's get started

We will perform a very simple function and get a flavour of the similarities and differences to working in `R`.

`date` is a command that prints out the date and time. Copy and paste this command into your terminal

```
date
```

This prints out the date/time in UTC

More information on using the date function (here)[https://www.geeksforgeeks.org/date-command-linux-examples/]

We can also ask for the output for a particular timezone using the `TZ` function and `date`

```
TZ=Europe/London date
```

Or we can ask the computer what the date will be next Tuesday…

```
date --date="next tue"
```

2.5.1 Downloading data

We will start by typing in an instruction to download data from an online data repository, unpack the contents and inspect it:

- curl is a command line tool for transferring data to and from the server here we will use this to download data from an online repository.
- tar will *unpack* the data from a compressed file format
- cd change the directory so we *land* in the new folder we have made

```
curl -L -o unix_intro.tar.gz https://ndownloader.figshare.com/files/15573746
tar -xzvf unix_intro.tar.gz && rm unix_intro.tar.gz
cd unix_intro
```

Check each command line has run, in the example above you might find that the first two lines run, to download and unpack data, while the last line to change directory doesn't run until you hit enter

2.5.2 More functions

Unlike date, most commands require arguments and won't work without them. head is a command that prints the first lines of a file, so it requires us to provide the file we want it to act on:

```
head example.txt
```

Here “example.txt” is the required argument, and in this case it is also what’s known as a positional argument. Whether things need to be provided as positional arguments or not depends on how the command or program we are using was written.

Sometimes we need to specify the input file by putting something in front of it (e.g. some commands will use the -i flag, but it’s often other things as well).

Q. What’s in the text file? - Click here for Answer

Pretty boring, each line contains the text “This is line” followed by the line number e.g.

- *This is line 1*

- *This is line 2*

etc.

There are also optional arguments for the head command. The default for head is to print the first 10 lines of a file. We can change that by specifying the -n flag, followed by how many lines we want:

```
head -n 5 example.txt
```

How would we know we needed the -n flag for that? There are a few ways to find out. Many standard Unix commands and other programs will have built-in help menus that we can access by providing –help as the only argument:

```
head --help
```

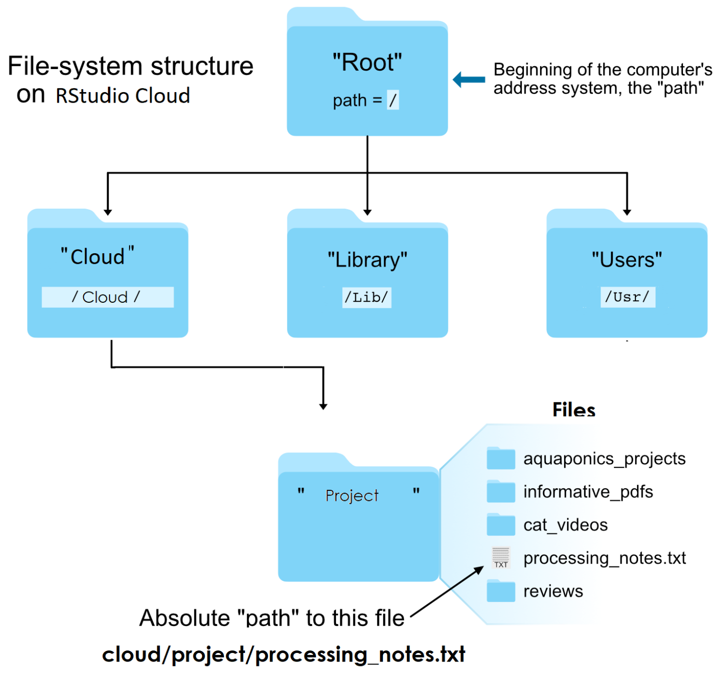
Again this is very similar to the logic in which R commands are strucutred e.g. `??ggplot()` The synatx is similar even if the specific icons or arguments are different.

Remember just like with R, one of your best friends is Google! As you get familiar with any language or OS we might remember a few flags or specific options, but searching for options and details when needed is definitely the norm!

2.6 Unix File Structure

There are two special locations in all Unix-based systems: the “root” location and the current user’s “home” location. “Root” is where the address system of the computer starts; “home” is usally where the current user’s location starts.

Just to be awkward RStudio Cloud actually has us working in a different location “Cloud”, which is underneath Root but separate to home which would be in the “Users” folder.



We tell the command line where files and directories are located by providing their address, their "path". If we use the pwd command (for print working directory), we can find out what the path is for the directory we are sitting in.

```
pwd
```

And if we use the ls command (for list), we can see what directories and files are in the current directory we are sitting in.

```
ls
```

Note

Why is it important to know this? Usually when you are working on a Unix-like environment there is no GUI (nice-click and point interface), all commands have to be submitted through the terminal. So you would have to get used to navigating directories with typed commands, and it's useful to know what the standard hierarchy is.

2.7 Absolute vs relative file paths

You should be used to these concepts from your work with R projects.

There are two ways to specify the path (address) of the file we want to do something to:

- An **absolute path** is an address that starts from an explicitly specified location: usually the "root" `/` or the "home" `~/` location. (Side note, because we also may see or hear the term, the "full path", is usually the absolute path that starts from the "root" /.)
- A **relative path** is an address that starts from wherever we are currently sitting (the working directory). For example, let's look again at the head command we ran above:

```
    head example.txt
```

What we are actually doing here is using a relative path to specify where the "example.txt" file is located. This is because the command line **automatically looks** in the current working directory if we don't specify anything else about its location.

We can also run the same command on the same file using an **absolute path** - note Rstudio cloud has a slightly unique set-up in that we start from a folder designated cloud:

```
    head /cloud/project/unix_intro/example.txt
```

The previous two commands both point to the same file right now. But the first way, head example.txt, will only work if we are entering it while "sitting" in the directory that holds that file, while the second way will work no matter where we happen to be in the computer.

It is **important to always think about where** we are in the computer when working at the command line. One of the most common errors/easiest mistakes to make is trying to do something to a file that isn't where we think it is. Let's run head on the "example.txt" file again, and then let's try it on another file: "notes.txt":

```
head example.txt
```

```
head notes.txt
```

Here the head command works fine on "example.txt", but we get an error message when we call it on "notes.txt" telling us no such file or directory. If we run the ls command to list the contents of the current working directory, we can see the computer is absolutely right – spoiler alert: it usually is – and there is no file here named "notes.txt".

The ls command by default operates on the current working directory if we don't specify any location, but we can tell it to list the contents of a different directory by providing it as a positional argument:

```
ls
```

```
ls experiment
```

We can see the file we were looking for is located in the subdirectory called "experiment". Here is how we can run head on "notes.txt" by specifying an accurate relative path to that file:

```
head experiment/notes.txt
```

2.8 Moving around

We can also move into the directory containing the file we want to work with by using the `cd` command (**c**hange **d**irectory). This command takes a positional argument that is the path (address) of the directory we want to change into. This can be a relative path or an absolute path. Here we'll use the relative path of the subdirectory, "experiment", to change into it

```
cd experiment/
```

```
pwd
```

```
ls
```

```
head notes.txt
```

Great. But now how do we get **back "up"** to the directory above us? One way would be to provide an absolute path, like `cd /cloud/project/unix_intro`, but there is also a handy shortcut. `..` which are special characters that act as a relative path specifying "up" one level – one directory – from wherever we currently are.

So we can provide that as the positional argument to cd to get back to where we started:

```
cd ..
```

Moving around the computer like this might feel a bit cumbersome and frustrating at first, but after spending a little time with it, you will get used to it, and it starts to feel more natural.

Note

One way to speed things up is to start using **tab** to perform **tab-completion** often this will auto-complete file names! Press tab twice quickly and it will print all possible combinations.

2.9 Summary

While maybe not all that exciting, these things really are the foundation needed to start utilizing the command line – which then gives us the capability to use lots of tools that only work at a command line, manipulate large files rapidly, access and work with remote computers, and more! These are the fundamental tools that every scientist needs to work with **big data**.

2.9.1 Terms

Term	**What it is**
path	the address system the computer uses to keep track of files and directories
root	where the address system of the computer starts, /
home	where the current user's location starts, ~/
absolute path	an address that starts from a specified location, i.e. root, or home
relative path	an address that starts from wherever we are
tab-completion	our best friend

2.9.2 Commands

Command	**What it is**
date	prints out information about the current date and time
head	prints out the first lines of a file
pwd	prints out where we are in the computer (print working directory)
ls	lists contents of a directory (list)
cd	change directories

2.9.3 Special characters

Command	**What it is**
Characters	Meaning
/	the computer's root location
~/	the user's home location
../	specifies a directory one level "above" the current working directory

2.10 Summary

You won't get used to operating in bash, or moving around directories using just the command line in a single session. So if you think you are interested in developing your bioinformatic skills, spend some time practising.

Here is a link to a couple of extended tutorials you can bookmark if you want to explore this further:

https://datacarpentry.org/shell-genomics/01-introduction/index.html

2.11 Stretch yourself - optional extras to try a couple of other skills

2.11.1 Creation

I want to create a new directory to store some code files I'm going to write later, so I'll use `mkdir` to create a new directory called Code:

Check you are in the `unix_intro` folder - Click here for Answer

```
pwd
```

Make a new directory called Code - Click here for Answer

```
mkdir Code
```

Check this folder has been created using a list function

```
ls
```

Note that I used a relative file path to create the Code directory - but I could have also specified an absolute filepath to generate that folder in whatever location I want.

There are a few ways to make new files on the command line. The simplest is to generate a blank file with the `touch` command followed by the path (relative or absolute) to the file you want to create

Make a new file called data-science-class.txt - Click here for Answer

```
touch data-science-class.txt
ls -l
```

Note here I could just use ls to list all files and folders in a directory, but if i set the flag `-l` then it will produce a `l`ong list of files.

If the entry in the first column is a `d`, then the row in the table corresponds to a directory, otherwise the information in the row corresponds to a file.

The string of characters following the `d` in the case of a directory or following the first `-` in the case of a file represent the permissions for that file or directory - I won't cover that here - but some of the links I provide go into more detail.

Chapter 3

NGS sequence analysis

3.1 Background on high throughput sequencing

High-throughput sequencing, also known as massively parallel sequencing or next-generation sequencing (NGS), is a collection of methods and technologies that can sequence DNA thousands/millions of fragments at a time. The market leader on NGS is Illumina, and an overview of their technology is in the video below.

There are many uses for high throughput sequencing including:

- Whole genome sequencing
- Amplicon sequencing - PCR of a targeted gene(s) is step one
  - environmental DNA
  - 16S Bacterial community analysis
  - Targeted gene panels
- RNA sequencing
- ChIP sequencing: Protein-DNA interaction analysis

Importantly a **lot** of the basic bioinformatics is the same across these technologies, because the data that is produced from the sequencing run is basically the same as well. The *big* data generated here are all massive *FASTQ* files, processing these follows basically the same initial pipeline for all applications.

3.2 Some terms

Term	**What it is**
Insert	the DNA fragment that is being used for sequencing

Read	The part of the insert that is sequenced
Single read	A procedure in which the insert is sequenced once
Paired end	A procedure in which the insert is sequenced twice, once from each en
Flowcell	A small glass chip on which the DNA fragments are attached and the
Lane	The flowcell has 8 physically separate lanes. Sequencing occurs in par
Multiplexing/Demultiplexing	Sequencing multiple independent biological samples on the same lane
Pipeline	The series of computational processes used to go from FASTQ data fi

3.3 The data

This data comes from exploring an underwater mountain ~3 km down at the bottom of the Pacific Ocean that serves as a low-temperature (~5-10°C) hydrothermal venting site.

This amplicon dataset was generated from DNA extracted from crushed basalts collected from across the mountain with the goal being to begin characterizing the microbial communities of these deep-sea rocks. No one had ever been here before, so this was a broad-level community survey. The sequencing was done on the Illumina MiSeq platform with 2x300 paired-end sequencing using primers targeting the V4 region of the 16S rRNA gene.

There are 20 samples total: 4 extraction "blanks" (nothing added to DNA extraction kit), 2 bottom-water samples, 13 rocks, and one biofilm scraped off of a rock. None of these details are important for you to remember, it's just to give some overview if you care.

Q. Why would we include "blank" samples in our sequencing run? - Click here for Answer

This sort of "environmental data" is very at risk of contamination, although the DNA extractions, and PCRs have to be run under sterile conditions or they will pick up bacteria from the lab and not the sample. Despite our best efforts we can still get minor contamination, these "blank" runs can be useful as anything in these samples **cannot** *have come from our deep-sea rocks, and therefore we could choose to "remove" sequences that match these in our other samples and label them as contamination.*

In the following figure, overlain on the map are the rock sample collection locations, and the panes on the right show examples of the 3 distinct types of rocks collected: 1) basalts with highly altered, thick outer rinds (>1 cm); 2) basalts that were smooth, glassy, thin exteriors (~1-2 mm); and 3) one calcified carbonate.

Altogether the uncompressed size of the working directory we are downloading here is ~300MB - this is about 10% of the full dataset - we are using a reduced dataset to minimise system requirements and speed up the workflow.

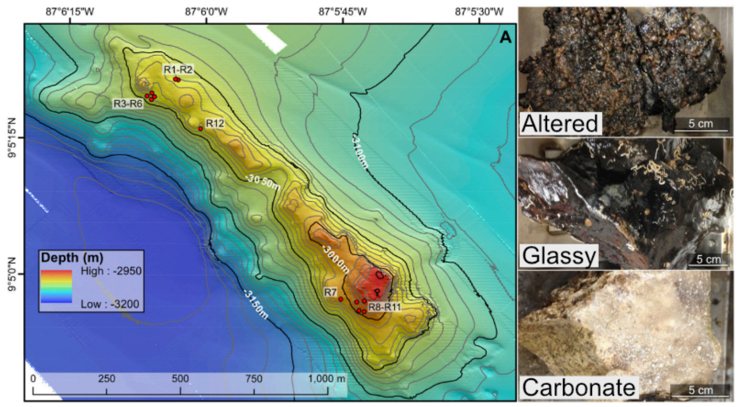


Figure 3.1: Map of collection sites and examples of the rocks collected

To get started, be sure you are in the "Terminal" window. We will be working here for the first step of importing the data, and removing the primers from our data. We can import our data using the `curl` function, we will then remove the primers using a program called `cutadapt` which is written in Python.

Make sure when you open the terminal you are in the project directory (and refer to last weeks notes if you need to check how to do this).

Don't switch over to R (the "Console" tab in the Binder/RStudio environment) until noted. You can download the required dataset and files by copying and pasting the following commands into your command-line terminal:

```
curl -L -o dada2_amplicon_ex_workflow.tar.gz https://ndownloader.figshare.com/files/23066516
tar -xzvf dada2_amplicon_ex_workflow.tar.gz
rm dada2_amplicon_ex_workflow.tar.gz
cd dada2_amplicon_ex_workflow/
```

Q. Can you work out what each of these lines of code might be doing? - Click here for Answer

In brief these commmands:

- download/curl some external data - uncompress into a folder - remove the compressed file - change the working directory to the newly created folder

In our working directory there are now 20 samples with one forward (R1) and one reverse (R2) read each, each file has DNA sequences with per-base-call quality information, for a total of 40 fastq files (.fq). It is a good idea to have a file with all the sample names to use for various things throughout, so here's making that file based on how these sample names are formatted.

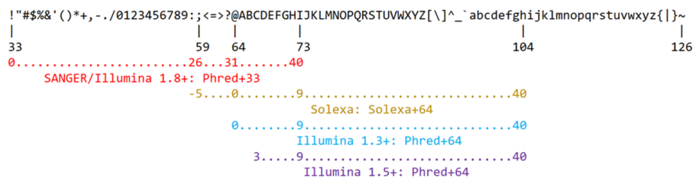
```
ls *_R1.fq | cut -f1 -d "_" > samples
```

3.3.1 FASTQ?

FASTQ format is a text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores. As each nucleotide in a *read* is sequenced, it is assigned a *Phred quality score*. This score is the assigned *probability* of the sequencer having made an incorrect base call

Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.90%
40	1 in 10,000	99.99%
50	1 in 100,000	100.00%
60	1 in 1,000,000	100.00%

These quality scores are stored within the FASTQ files as ASCII characters



```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
|                         |    |        |                              |                     |
33                        59   64       73                             104                   126
0........................26...31.......40
     SANGER/Illumina 1.8+: Phred+33
                          -5....0........9.............................40
                                          Solexa: Solexa+64
                               0........9.............................40
                                        Illumina 1.3+: Phred+64
                                  3.....9.............................40
                                         Illumina 1.5+: Phred+64
```

Figure 3.2: Phred quality scores as ASCII characters

This is all stored together as four simple lines of repeating text so that a FASTQ file containing a single sequence might look like this:

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

A single FASTQ file may contain millions of sequencing reads. Let's look at the first 40 lines of *one* of our FASTQ files. And check it looks like a standard format.

```
head -40 B1_sub_F1.fq
```

Q. By eye, can you tell whether these few lines look to be of good quality? - Click here for Answer

The ASCII characters are repeated every fourth line, most of these reads appear to be G letters or close to this - indicating greater than 99.9% accuracy - pretty good. Very observant students might have noticed that the end of the reads appear to be of lower quality. More on this later

3.4 The Pipeline

This is a very simple overview of the pipeline we will run, some of these steps (especially early ones - are applicable to lots of NGS applications), later on they become more specific to *our* data.

- Import the FASTQ files and demultiplex (this step was done for us)
- Remove adapters and primers (these may be included with our reads, but they are not part of the *natural* DNA sequence)
- Check FASTQ data quality and trim/filter reads accordingly
- Dereplicate (collapse identical sequences and choose a representative)
- Assign ASVs - decide if (non-identical) sequences are similar enough to be considered as from the same species
- Join forward and reverse reads together
- Assign ASVs to Taxonomies
- Count the abundance of different ASVs
- Export taxonomy file, ASV fasta sequence file and count file to R for analysis

3.5 Removing Primers

To start, we need to remove the primers from all of these (the primers used for this run are in the "primers.fa" file in our working directory), and here we're going to use `cutadapt` to do that at the command line ("Terminal" tab).

First we need to install `cutadapt`

```
python3 -m pip install --user --upgrade cutadapt
```

Note

You will probably get a warning message about PATH. You can ignore this, what it means is that CUTADAPT has been installed in your home directory, in order to use it we need to specify the absolute path TO cutadapt when we call it. This is done for you in the next code box below.

Cutadapt operates on one sample at at time, so we're going to use a wonderful little bash *loop* to run it on all of our samples.

3.5.1 Loops

Loops are extremely powerful way of controlling iteration. We can specify that a line of code is repeated across multiple objects. In this example we use the `samples` file we made earlier as the list of files across which we want this function of removing primers to loop. These same lines will then repeat until all the specified iterations are complete.

We won't break down exactly how this loop works - but they are used across all programming languages (including R) and you can check out the R4DS book for an introduction to building your own loops (and custom functions!) here.

For now just copy and paste this code exactly into the Terminal.

```
for sample in $(cat samples)
do

    echo "On sample: $sample"

    ~/.local/bin/cutadapt -a ^GTGCCAGCMGCCGCGGTAA...ATTAGAWACCCBDGTAGTCC \
    -A ^GGACTACHVGGGTWTCTAAT...TTACCGCGGCKGCTGGCAC \
    -m 215 -M 285 --discard-untrimmed \
    -o ${sample}_sub_R1_trimmed.fq.gz -p ${sample}_sub_R2_trimmed.fq.gz \
    ${sample}_sub_R1.fq ${sample}_sub_R2.fq \
    >> cutadapt_primer_trimming_stats.txt 2>&1

done
```

Here's a before-and-after of one of our files - if you look at the sequences supplied: GTGCCAGCMGCCGCGGTAA...ATTAGAWACCCBDGTAGTCC these indicate the forward primer and the reverse primer (our amplicon is everything inbetween). If you look at our before and after you should see these were at the start and end of the sequence but have now been trimmed off.

```
### R1 BEFORE TRIMMING PRIMERS
head -n 2 B1_sub_R1.fq
# @M02542:42:000000000-ABVHU:1:1101:8823:2303 1:N:0:3
# GTGCCAGCAGCCGCGGTAATACGTAGGGTGCGAGCGTTAATCGGAATTACTGGGCGTAAAGCGTGCGCAGGCGGTCTTGT
# AAGACAGAGGTGAAATCCCTGGGCTCAACCTAGGAATGGCCTTTGTGACTGCAAGGCTGGAGTGCGGCAGAGGGGGATGG
# AATTCCGCGTGTAGCAGTGAAATGCGTAGATATGCGGAGGAACACCGATGGCGAAGGCAGTCCCCTGGGCCTGCACTGAC
# GCTCATGCACGAAAGCGTGGGGAGCAAACAGGATTAGATACCCGGGTAGTCC
```

```
### R1 AFTER TRIMMING PRIMERS
head -n 2 B1_sub_R1_trimmed.fq
# @M02542:42:000000000-ABVHU:1:1101:8823:2303 1:N:0:3
# TACGTAGGGTGCGAGCGTTAATCGGAATTACTGGGCGTAAAGCGTGCGCAGGCGGTCTTGTAAGACAGAGGTGAAATCCC
# TGGGCTCAACCTAGGAATGGCCTTTGTGACTGCAAGGCTGGAGTGCGGCAGAGGGGGATGGAATTCCGCGTGTAGCAGTG
# AAATGCGTAGATATGCGGAGGAACACCGATGGCGAAGGCAGTCCCCTGGGCCTGCACTGACGCTCATGCACGAAAGCGTG
# GGGAGCAAACAGG
```

You can look through the output of the cutadapt stats file we made ("cutadapt_primer_trimming_stats.txt") to get an idea of how things went.

Here's a little one-liner to look at what fraction of reads were retained in each sample (column 2) and what fraction of bps were retained in each sample (column 3):

```
    paste samples <(grep "passing" cutadapt_primer_trimming_stats.txt | cut -f3 -d "(" | tr -d ")
```

```
# B1    96.5%   83.0%
# B2    96.6%   83.3%
# B3    95.4%   82.4%
# B4    96.8%   83.4%
# BW1   96.4%   83.0%
# BW2   94.6%   81.6%
# R10   92.4%   79.8%
# R11BF 90.6%   78.2%
# R11   93.3%   80.6%
# R12   94.3%   81.4%
# R1A   93.3%   80.5%
# R1B   94.0%   81.1%
# R2    94.0%   81.2%
# R3    93.8%   81.0%
# R4    95.5%   82.4%
# R5    93.7%   80.9%
# R6    92.7%   80.1%
# R7    94.4%   81.5%
# R8    93.2%   80.4%
# R9    92.4%   79.7%
```

This looks like it worked pretty well! Some reads were discarded entirely `-m 215 -M 285 --discard-untrimmed \` anything <215bp or >285bp was discarded. Looks like c. 6-8% We lost a greater proportion of bp overall, but this is the program working as it should, making most of our reads a little shorter as it cuts off the primers. Overall it looks like we lost c.20% of bp.

Importantly all of our files have behaved *roughly* the same.

With primers removed, we're now ready to switch R and start using DADA2!

3.6 DADA2

Switch from the Terminal to Console now. We are working in R for the rest of this workflow

```
  library(dada2)

  setwd("dada2_amplicon_ex_workflow")

  list.files() # make sure what we think is here is actually here

  ## first we're setting a few R objects we're going to use ##
    # one with all sample names, by scanning our "samples" file we made earlier
samples <- scan("samples", what="character")

    # one holding the file names of all the forward reads
forward_reads <- paste0(samples, "_sub_R1_trimmed.fq.gz")

    # and one with the reverse
reverse_reads <- paste0(samples, "_sub_R2_trimmed.fq.gz")

    # and variables holding file names for the forward and reverse
    # filtered reads we're going to generate below
filtered_forward_reads <- paste0(samples, "_sub_R1_filtered.fq.gz")
filtered_reverse_reads <- paste0(samples, "_sub_R2_filtered.fq.gz")
```

3.7 Quality trimming/filtering

We did a filtering step above with cutadapt (where we eliminated reads that had imperfect or missing primers and those that were shorter than 215 bps or longer than 285), but in DADA2 we'll implement a trimming step as well (where we trim reads down based on some quality threshold rather than throwing the read away).

Since we're potentially shortening reads further, we're again going to include another minimum-length filtering component. We can also take advantage of a handy quality plotting function that DADA2 provides to visualize how you're reads are doing, plotQualityProfile().

By running that on our variables that hold all of our forward and reverse read filenames, we can easily generate plots for all samples or for a subset of them. So let's take a peak at that to help decide our trimming lengths: It's good to try to keep a bird's-eye view of what's going on. So here is an overview of the main processing steps we'll be performing with cutadapt and DADA2. Don't worry if anything seems unclear right now, we will discuss each at each step.

```
plotQualityProfile(forward_reads[17:20])

plotQualityProfile(reverse_reads[17:20])
```

All forwards look pretty similar to each other, and all reverses look pretty similar to each other, but worse than the forwards, which is common – the Illumina sequencer reads all of the molecules in the forward orientation *first*, then the clusters are flipped and read in reverse. But this means a lot of the chemical reagents start to get used up or degraded, so it is usual to get lower quality reverse reads.

On the plots produced

- the x axis is the nucleotide bases starting from the beginning of the read moving to the end
- the y axis is the average quality score for the base in that position
- the green line is the median quality score of the base at that position
- the orange lines are quartiles

These quality profiles are based entirely on taking the *average* PHRED scores for sequences at that position in the sample

Here, I'm going to cut the forward reads at 250 and the reverse reads at 200 – roughly where both sets maintain a median quality of 30 or above – and then see how things look. But we also want to set a minimum length to filter out truncated sequences, so we will set a minimum acceptable read length of 175bp (any reads shorter than this will be discarded).

In DADA2, this quality-filtering step is done with the `filterAndTrim()` function:

```
filtered_out <- filterAndTrim(forward_reads, filtered_forward_reads,
                reverse_reads, filtered_reverse_reads, maxEE=c(2,2),
                rm.phix=TRUE, minLen=175, truncLen=c(250,200))
```

This function made a bunch of output files "filtered_forward_reads" and "filtered_reverse_reads" we can see these in our project pane. Or if we were working on a server without a GUI we could use `list.files()` in R or `ls` in our Terminal.

We also generated an R file called filtered_out. This is a simple matrix holding how many reads went *in* for each file and how many came *out*.

Check it in R.

```
    filtered_out
```

We can take a look at the filtered reads visually - we expect to have trimmed off that section where quality drops

```
    plotQualityProfile(filtered_reverse_reads[17:20])
```

Looking Good!

3.8 Dereplication

Dereplication is a common step in many amplicon processing workflows. Instead of keeping 100 identical sequences and doing all downstream processing to all 100 -costing computer processing power and time, you can keep/process just one of them, and just attach the number x100 to it. Now this acts as a representative for 100 identical sequences.

```
derep_forward <- derepFastq(filtered_forward_reads, verbose=TRUE)
names(derep_forward) <- samples # the sample names in these objects are initially the
derep_reverse <- derepFastq(filtered_reverse_reads, verbose=TRUE)
names(derep_reverse) <- samples
```

3.9 ASV's

This is where we start to take our raw sequence data and infer *true* biological sequences. It uses an algorithm to look at the consensus quality score and abundance for each *unique* sequence. It then determines whether this sequence is more likely to be of biological origin or a spurious sequencing error.

> **Note**
>
> This step may take a few minutes to run, so be patient!

```
load("amplicon_dada2_ex.RData")

dada_forward <- dada(derep_forward, err=err_forward_reads, pool="pseudo")

dada_reverse <- dada(derep_reverse, err=err_reverse_reads, pool="pseudo")
```

3.10 Merging reads

Now DADA2 merges the forward and reverse ASVs to reconstruct our full target amplicon requiring the overlapping region to be identical between the two. By default it requires that at least 12 bps overlap, but in our case the overlap should be much greater. If you remember above we trimmed the forward reads to 250 and the reverse to 200, and our primers were 515f–806r. After cutting off the primers we're expecting a typical amplicon size of around 260 bases, so our typical overlap should be up around 190. That's estimated based on E. coli 16S rRNA gene positions and very back-of-the-envelope-esque of course, so to allow for true biological variation and such I'm going ot set the minimum overlap for this dataset for 170. I'm also setting the trimOverhang option to TRUE in case any of our reads go passed their opposite primers (which I wouldn't expect based on our trimming, but is possible due to the region and sequencing method).

```
merged_amplicons <- mergePairs(dada_forward, derep_forward, dada_reverse,
                    derep_reverse, trimOverhang=TRUE, minOverlap=170)

  # this object holds a lot of information that may be the first place you'd want to look if you
class(merged_amplicons) # list
length(merged_amplicons) # 20 elements in this list, one for each of our samples
names(merged_amplicons) # the names() function gives us the name of each element of the list

class(merged_amplicons$B1) # each element of the list is a dataframe that can be accessed and mar

names(merged_amplicons$B1) # the names() function on a dataframe gives you the column names
# "sequence"  "abundance" "forward"   "reverse"   "nmatch"    "nmismatch" "nindel"    "prefer"
```

3.11 Count table

Now we can generate a count table with the makeSequenceTable() function. This is one of the main outputs from processing an amplicon dataset. You may have also heard this referred to as a biome table, or an OTU matrix.

```
seqtab <- makeSequenceTable(merged_amplicons)
class(seqtab) # matrix
dim(seqtab) # 20 2521
```

3.12 Overview

The developers' DADA2 tutorial provides an example of a nice, quick way to pull out how many reads were dropped at various points of the pipeline. This can serve as a jumping off point if you're left with too few sequences at the end to help point you towards where you should start digging into where they are being dropped. Here's a slightly modified version:

```
  # set a little function
getN <- function(x) sum(getUniques(x))

  # making a little table
summary_tab <- data.frame(row.names=samples, dada2_input=filtered_out[,1],
               filtered=filtered_out[,2], dada_f=sapply(dada_forward, getN),
               dada_r=sapply(dada_reverse, getN), merged=sapply(merged_amplicons, getN
               nonchim=rowSums(seqtab.nochim),
               final_perc_reads_retained=round(rowSums(seqtab.nochim)/filtered_out[,1]

summary_tab
```

And it might be useful to write this table out of R, saving it as a regular file

```
write.table(summary_tab, "read-count-tracking.tsv", quote=FALSE, sep="\t", col.names=N
```

3.13 Assign taxonomy

Running the Taxonomy assignment step below can take anywhere from 30 minutes to a few hours depending on how much RAM we provide. So for this example run - we will skip this step and load an R.data file which has this information in it already

```
load("amplicon_dada2_ex.RData")
```

Example code for running taxonomy assignment - Click here

So we won't run this code in this example, but here it is for reference.

```
## downloading DECIPHER-formatted SILVA v138 reference
download.file(url="http://www2.decipher.codes/Classification/TrainingSets/SILVA_SSU_r1

## loading reference taxonomy object
load("SILVA_SSU_r138_2019.RData")

## loading DECIPHER
library(DECIPHER)

## creating DNAStringSet object of our ASVs
```

```
dna <- DNAStringSet(getSequences(seqtab.nochim))

## and classifying
tax_info <- IdTaxa(test=dna, trainingSet=trainingSet, strand="both", processors=NULL)
```

3.14 Standard goods

The typical standard outputs from amplicon processing are

- a fasta file: each ASV represented by a sequence `asv_fasta_no_contam`
- a count table: how many sequences of each ASV in each sample `asv_tab_no_contam`
- a taxonomy file: the closest biological species to the fasta sequence `asv_tax_no_contam`

These objects from DADA2 can then be analysed to start to understand the different bacterial communities from our deep-sea study:

Note

- These three files are relatively small simply lists now, you can type them into the R console and inspect these outputs if you wish. Do they make sense to you?

3.15 Functions list

Command	**What it is**
cutadapt/filterAndTrim()	remove primers and quality trim/filter
learnErrors()	generate an error model of our data
derepFastq	dereplicate sequences
dada()	infer ASVs on both forward and reverse reads independently
mergePairs()	merge forward and reverse reads to further refine ASVs
makeSequenceTable()	generate a count table
removeBimeraDenovo()	screen for and remove chimeras
IdTaxa()	assign taxonomy

3.16 Summary

We have imported FASTQ data from an Illumina sequencing run, processed the files to remove poor quality reads and trim primers. We have then put this through a microbiome specific bioinformatics pipeline to assign millions of individual reads to more manageable representative sequences. We have assigned taxonomies to these sequences and tallied them, so that **next time** we can actually inspect our data and start to make visuals that describe our microbial communities.

Chapter 4

Analysis in R - UNDER CONSTRUCTION

```
library(phyloseq)
library(vegan)
library(DESeq2)
library(tidyverse)
library(dendextend)
library(viridis)
library("reshape")
```

4.1 Important files

We're mostly going to be working with just 3 files now.

- A count table: the number of reads for each unique sequence
- A taxonomy table: the assigned taxonomy for each sequence according to the SILVA database
- A sample file: this is the "metadata" it contains any information *we* have provided about the different samples

  Note

  Since we've already used decontam to remove likely contaminants, we're dropping the "blank" samples from our count table which in the table we're reading in are the first 4 columns. That's what's being done by the [, -c(1:4)] part at the end there.

```
sample_info_tab <- read.table("sample_info.tsv", header=T, row.names=1,
                    check.names=F, sep="\t")
```

```
ps <- phyloseq(otu_table(asv_tab_no_contam, taxa_are_rows=T),
               sample_data(sample_info_tab),
               tax_table(asv_tax_no_contam))
```

Make graphs

```
 ps %>%
  tax_glom(taxrank="class") %>%
  transform_sample_counts(function(x){x/sum(x)})%>%
  psmelt() %>%
  filter(Abundance >0.05)
```

```
 ps %>%
  tax_glom(taxrank="class") %>%
  transform_sample_counts(function(x){x/sum(x)})%>%
  psmelt() %>%
  filter(Abundance >0.05)%>%
  ggplot(aes(x=Sample, y=Abundance, colour=class))+
  geom_bar(stat="identity")
```

sample_info_tab$Sample <- rownames(sample_info_tab)

ps_richness$Sample <- rownames(ps_richness)

richness -> ps %>% estimate_richness()

left_join(sample_info_tab, ps_richness, "Sample")%>% ggplot(aes(x=char, y=Simpson))+geom_boxplot

Heatmap

ps_rare_top20 <- prune_taxa(names(sort(taxa_sums(ps),TRUE)[1:20]), ps)

plot_heatmap(ps_rare_top20,"NMDS",distance = "bray",sample.label="char")

Ordination

ord1<-plot_ordination(ps, ord.nmds.bray, color="char", title="Bray NMDS")
> > #Plot with Ellipses assuming normal distribution
> ord1 + stat_ellipse(type="norm") + theme_bw()

Answer some questions?

1) alpha diversity differs between ‘Early’ and ‘Late’ sampled mouse gut microbiomes
2) beta diversity differs between ‘Early’ and ‘Late’ sampled mouse gut microbiomes

4.2 Summary

Chapter 5

Applications

Some *significant* applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

Chapter 6

Final Words

We have finished a nice book.