2025 Coriell Bioinformatics Internship

Gennaro Calendo

2025-05-20

Table of contents

Bioinformatics Notes

This book contains assorted bioinformatics related notes. It contains some workflows for RNA-seq, ATAC-seq, and methylation analysis, as well as some notes describing best practices for structuring data projects, and presenting science. We hope to keep this book updated as more useful topics come up in the lab.

‘Best’ practices for data projects

Science requires reproducibility, not only to ensure that your results are generalizable but also to make your life easier. One overlooked aspect of of learning data science is creating consistent and clear organization of your projects and learning how to keep your data safe and easily searchable.

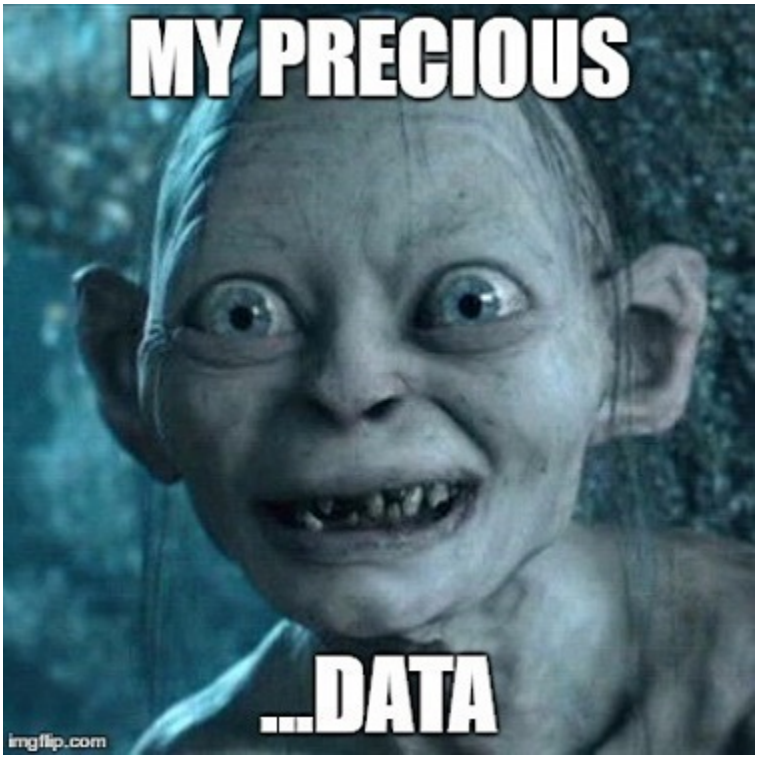
Quick note (TODO)

Citations incoming. Please note that much of this document does not contain original material. References are listed at the end of the chapter (I still need to get my .bib files in order to give proper attribution).

Why care about data management?

Computing is now an essential part of research. This is outlined beautifully in the paper, “All biology is computational biology” by Florian Markowetz. Data is getting bigger and bigger and we need to be equipped with the tools for storing, manipulating, and communicating insights derived from it. However, most researchers are never taught good computational practices. Computational best practices are imperitive. Implementing best (or good enough) practices can improve reproducibility, ensure correctness, and increase efficiency.

File Management



File types and file names

As a data scientist you'll be dealing with a lot of files but have you ever considered *what* a file is? Files come in all shapes and formats. Some are very application specific and require specialized programs to open. For example, consider DICOM files that are used to store and manipulate radiology data. Luckily, in bioinformatics we tend to deal mainly with simple, plain text files, most often. Plain text files are typically designed to be both human and machine-readable. If you have the choice of saving any data, you should know that some formats will make your life easier. Certain file formats like TXT, CSV, TSV, JSON, and YAML are standard plain text file formats that are easy to share and easy to open and manipulate. Because of this, you

should prefer to store your data in machine-readable formats. Avoid .xlsx files for storing data. Prefer TXT, CSV, TSV, JSON, YAML, and HDF5.

If you have very large text files then you can use compression utilities to save space. Most bioinformatics software is designed to work well with gzip compressed data. `gzip` is a relatively old compression format. You could also consider using `xz` as a means to compress your data - just know that `xz` compression is less supported across tools.

File naming

File naming is important but often overlooked. You want your files to be named logically and communicate their contents. You also want your files to be named in a way that a computer can easily read. For example, spaces in filenames are a royal pain when manipulating files on the command line.

To ensure filenames are computer friendly, don't use spaces in filenames. Use only letters, numbers, and "-" "_" and "." as delimiters. For example:

```
# Bad
data for jozef.txt

# Okay
data-for-jozef.txt

# Best
2023-11-09_repetitive-element-counts.txt
```

Quiz

I have a directory with the following files:

```
a.txt
b.txt
c.txt
file with spaces in the name.txt
some other file.txt
```

What does the following code return? (Expected: For each file in the directory print the filename)

```
for FILE in $(ls); do echo $FILE; done
```

Bash interprets every space as a new word!

```
a.txt
b.txt
c.txt
file
with
spaces
in
the
name.txt
some
other
file.txt
```

Pro-tip

A simple alternative here is to use a text document with the basenames the files you want to loop over and then loop over the lines of the file instead.

```
SAMPLES=sample-names.txt

for SAMPLE in $(cat $SAMPLES); do
  doStuff ${SAMPLE}.txt;
done
```

More on filenames

It cannot be stressed enough how important filenames can be for an analysis. To get the most out of your files and to avoid catastrpohic failures, you should stick to some basic principles for naming files. First, use consistent and unique identifiers across all files that you generate for an experiment. For example, if you're conducting a study that has both RNA-seq and ATAC-seq data performed on the same subjects, don't name the files from the RNA-seq experiment `subject1.fq.fz` and the files from the ATAC-seq experiment `control_subject1.fq.gz` if they refer to the same sample. For small projects, it's fairly easy to create consistent and unique IDs for each subject. For large projects unique random IDs can be used.

For example, the following filenames would be bad:

```
subject1-control_at_96hr1.txt
s1_ctl-at-4days_2.txt
s2TRT4d1.txt
sbj2_Treatment_4_Days_Replicate_2.txt
```

Instead, look at these filenames.

```
subject1_control_4days_rep1.txt
subject1_control_4days_rep2.txt
subject2_treatment_4days_rep1.txt
subject2_treatment_4days_rep2.txt
```

These are better. *Why* are they better? They are consistent. The delimiter is consistent between the words ("_") and each of the words represents something meaningful about the sample. These filenames also do not contain any spaces and can easily be parsed automatically.

File naming best practices also apply to naming executable scripts. The name of the file should describe the function of the script. For example,

```
01_align_with_STAR.sh
```

is better than simply naming the file

```
01_script.sh
```

Pro-tip

One easy way to create unique random IDs for a large project is to concatenate descriptions and take the SHA/MDA5 hashsum.

```
echo "subject1_control_4days_rep1" | sha256
# 57f458a294542b2ed6ac14ca64d3c8e4599eed7a

echo "subject1_control_4days_rep2" | shasum
# b6ea9d729e57cce68b37de390d56c542bc17dea6
```

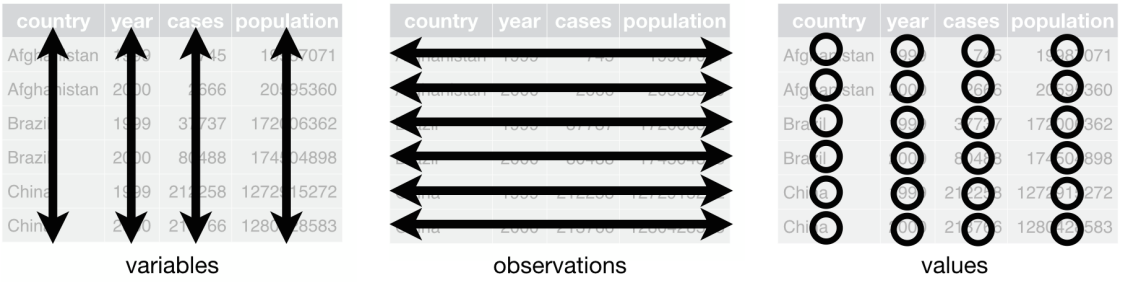
Create analysis friendly data - tidy data

The term tidy data was defined by Hadley Wickham to describe data which is amenable to downstream analysis. Most people are familiar with performing a quick and dirty data analysis in a program like Excel. You may have also used some of Excel's fancy features for coloring cells, adding bold and underlines to text, and formatting cells with other decorations. All of this tends to just be extra fluff. If you format you data properly then it will be much easier to perform downstream analysis on and will not require the use of extra decorations. This is true even in Excel!

To conform to the requirements of being *tidy*, the data should follow some simple principles:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

    a) Don't cram two variables into one value. e.g. "male_treated".
    b) Don't embed units into data. e.g. "3.4kg". Instead, put in column name e.g. "weight_kg" = 3.4



Once your data is in this format, it can easily be read into downstream programs like R, or parsed with command line text editing programs like `sed`.

The `iris` dataset in R provides a classic example of this format

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Pro-tip

`data.frame` like objects can be stored and retrieved efficiently using the Apache Arrow format instead of CSV files:

```
df <- arrow::read_parquet("path/to/file.parquet")
arrow::write_parquet(df, "path/to/different_file.parquet")
```

see R arrow for more details. There's also the nanoparquet package which provides a light weight reader/writer for parquet files.

Untidy(?) data

Some data formats are not amenable to the 'tidy' structure, i.e. they're just not best represented as long tables. For example, large/sparse matrices, geo-spatial data, R objects, etc.the lesson here is to store data in the format that is most appropriate for the data. For example, don't convert a matrix to a long format and save as a tsv file! Save it as an `.rds` file instead. Large matrices can also be efficiently stored as HDF5 files. Sparse matrices can be saved and accessed eficiently using the Matrix package in R. And if you are accessing the same data often, consider storing as a SQLite database and accessing with dbplyr or sqlalchemy in Python. The main point is don't force data into a format that you're familiar with only because you're familiar with that format. This will often lead to large file sizes and inefficient performance.

Save and lock all raw data

Keep raw data in its unedited form. This includes not making changes to filenames. In bioinformatics, it's common to get data from a sequencing facility with incomprehensible filenames. Don't fall victim to the temptation of changing these filenames! Instead, it's much better to keep the filenames exactly how they were sent to you and simply create a spreadsheet that maps the files to their metadata. In the case of a sample mix-up, it's much easier to make a change to a row in a spreadsheet then to track down all of the filenames that you changed and ensure they're correctly modified.

Once you have your raw data, you don't want the raw data to change in any way that is not documented by code. To ensure this, you can consider changing file permissions to make the file immutable (unchangable). Using bash, you can change file permissions with:

```
chattr +i myfile.txt
```

If you're using Excel for data analysis, lock the spreadsheet with the raw data and only make references to this sheet when performing calculations.

Large files

You'll probably be dealing with files on the order of 10s of GBs. You **do not** want to be copying these files from one place to another. This increases confusion and runs the risk of introducing errors. Instead avoid making copies of large local files or persistent databases and simply link to the files.

You can use use soft links. A powerful way of finding an linking files can be done with `find`

```
# Link all fastq files to a local directory
find /path/to/fq/files -name "*.fq.gz") -exec ln -s {} . \;
```

If using R, you can also sometimes specify a URL in place of a file path for certain functions.

```
# Avoid downloading a large GTF file - reads GTF directly into memory
url <- "https://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_human/release_44/gencode.v44.anno
gtf <- rtracklayer::import(url)
```

Backups

There are two types of people, those who do backups and those who will do backups.

The following are NOT backup solutions:

- Copies of data on the same disk
- Dropbox/Google Drive
- RAID arrays

All of these solutions mirror the data. Corruption or ransomware will propagate. For example, if you corrupt a file on your local computer and then push that change to DropBox then the file on DropBox is now also corrupted. I'm sure some of these cloud providers have version controlled files but it's better to just avoid the problem entirely by keeping good backups.

Use the 3-2-1 rule:

- Keep 3 copies of any important file: 1 primary and 2 backups.
- Keep the files on 2 different media types to protect against different types of hazards.
- Store 1 copy offsite (e.g., outside your home or business facility).

A backup is only a backup if you can restore the files!

Project Organization

Look familiar?

Project structure

One of the most useful changes that you can make to your workflow is the create a consistent folder structure for all of your analyses and stick with it. Coming up with a consistent and

generalizable structure can be challenging at first but some general guidelines are presented here and here

First of all, when beginning a new project, you should have some way of naming your projects. One good way of naming projects is to each project a descriptive name and append the date the project was started. For example, `brca_rnaseq_2023-11-09/` is better than `rnaseq_data/`. In six months when a collaborator wants their old BRCA data re-analyzed you'll thank yourself for timestamping the project folder and giving it a descriptive name.

My personal structure for every project looks like:

- Prefixing the project directory with the ISO date allows for easy sorting by date
- a README text file is present at the top level of the directory with a short description about the project and any notes or updates
- `data/` should contain soft links to any raw data or the results of downloading data from an external source
- `doc/` contains metadata documents about the samples or other metadata information about the experiment
- `results/` contains only data generated within the project. It has sub-directories for `figures/`, `data-files/` and `rds-files/`. If you have a longer or more complicated analysis then add sub-directories indicating which script generated the results.
- `scripts/` contains all analysis scripts numbered in their order of execution. Synchronize the script names with the results they produce.

A more complex example

Pro-tip

If using Rstudio, include an `.Rproj` file at the top level of your directory. Doing this enables you to use the here package to reference data within your project in a relative fashion. For example, you can more easily save data with:

```
plot_volcano(...)
ggsave(here("data", "results", "figures", "04", "volcano-plots.png"))
```

Tracking Changes

Record all steps used to generate the data

Always document all steps you used to generate the data that's present in your projects. This can be as simple as a README with some comments and a `wget` command or as complex as a snakemake workflow. The point is, be sure you can track down the exact source of every file that you created or downloaded.

For example, a README documenting the creation of the files needed to generate a reference index might look like:

```
Transcripts:
wget http://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_human/release_38/gencode.v38.transcri

Primary Assembly:
wget http://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_human/release_38/GRCh38.primary_asse

Create concatenated transcripts + genome for salmon (i.e. gentrome):
cat gencode.v38.transcripts.fa.gz GRCh38.primary_assembly.genome.fa.gz > gentrome.fa.gz

Create decoys file for salmon:
grep ">" <(gunzip -c GRCh38.primary_assembly.genome.fa.gz) | cut -d " " -f 1 > decoys.txt
sed -i.bak -e 's/>//g' decoys.txt
```

For more complicated steps include a script. e.g. creating a new genome index, subsetting BAM files, accessing data from NCBI, etc.

Pro-tip

A simple way to build a data pipeline that is surprisingly robust is just to create scripts for each step and number them in the order that they should be executed.

```
01_download.sh
02_process.py
03_makeFigures.R
```

You can also include a runner script that will execute all of the above. Or, for more consistent workflows, use a workflow manager like Nextflow, Snakemake, WDL, or good ole' GNU Make

Manual version control

Version control refers to the practice of tracking changes in files and data over their lifetime. You should always track any changes made to your project over the entire life of the project. This can be done either manually or using a dedicated version control system. If doing this manually, add a file called "CHANGELOG.md" in your docs/ directory and add detailed notes in reverse chronological order.

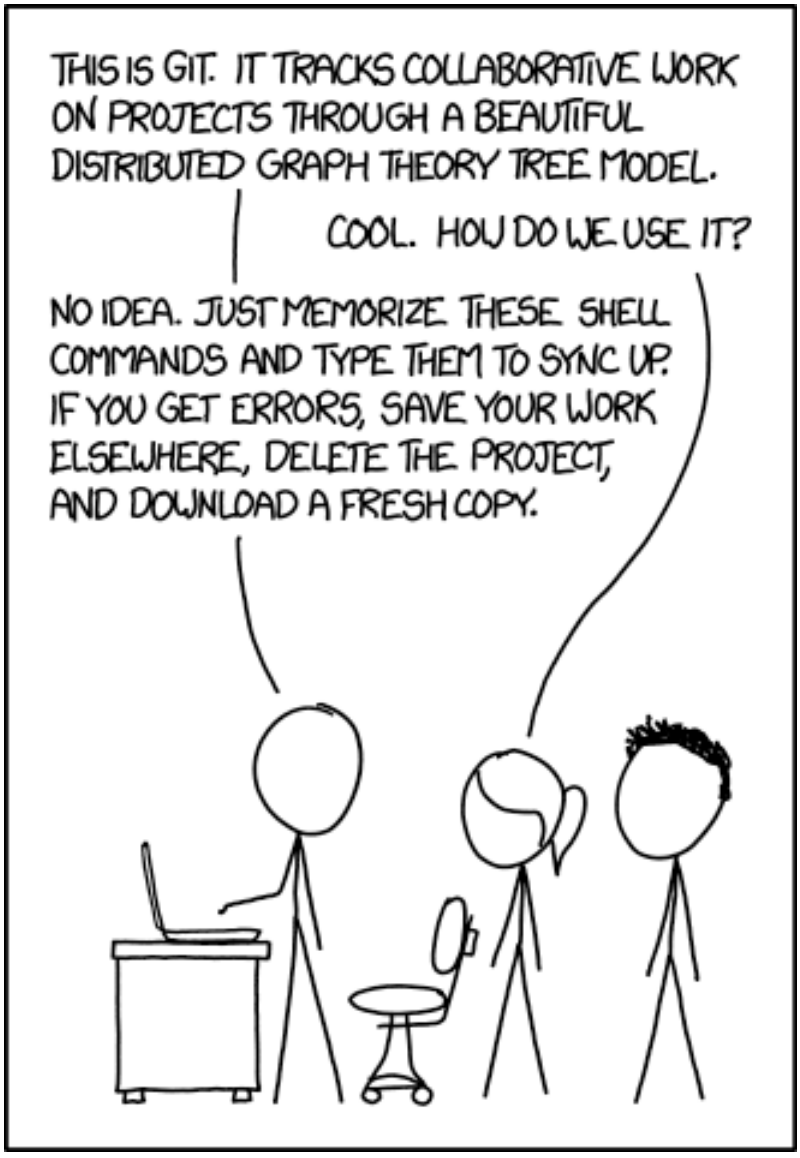
For example:

```
## 2016-04-08

* Switched to cubic interpolation as default.
* Moved question about family's TB history to end of questionnaire.

## 2016-04-06

* Added option for cubic interpolation.
* Removed question about staph exposure (can be inferred from blood test results).
```

If you make a significant change to the project, copy the whole directory, date it, and store it such that it will no longer be modified. Copies of these old projects can be compressed and saved with tar + xz compression

```
tar -cJvf old.20231109_myproject.tar.xz myproject/`
```

Version control with git



git is probably the *de facto* version control system in use today for tracking changes across software projects. You should strive to learn and use `git` to track your projects. Version control systems allow you to track all changes, comment on why changes were made, create parallel branches, and merge existing ones.

`git` is primarily used for source code files. Microsoft Office files and PDFs can be stored with Github but it's hard to track changes. Rely on Microsoft's "Track Changes" instead and save frequently.

It's not necessary to version control raw data (back it up!) since it shouldn't change. Likewise, backup intermediate data and version control the scripts that made it.

For a quick primer on Git and GitHub check out the book Happy Git with R or The Official GitHub Training Manual Anyone in the lab can join the coriell-research organization on Github and start tracking their code

Be careful committing sensitive information to GitHub

Software

Quick tips to improve your scripts

Place a description at the top of every script

The description should indicate who the author is. When the code was created. A short description of what the expected inputs and outputs are along with how to use the code. You three months from now will appreciate it when you need to revisit your analysis

For example:

```
#!/usr/bin/env python3
# Gennaro Calendo
# 2023-11-09
#
# This scripts performs background correction of all images in the
#  user supplied directory
#
# Usage ./correct-bg.py --input images/ --out_dir out_directory
#
from image_correction import background_correct

for img in images:
  img = background_correct(img)
  save_image(img, "out_directory/corrected.png")
```

Decompose programs into functions

Functions make it easier to reason about your code, spot errors, and make changes. This also follows the Don't Repeat Yourself principle aimed at reducing repetition by replacing it with abstractions that are more stable

Compare this chunk of code that rescales values using a min-max function (0-1)

```r
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

to this function which does the same thing

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df <- lapply(df, rescale01)
```

Which is easier to read? Which is easier to debug? Which is more efficient?

Give functions and variables meaningful names

- Programs are written for people and then computers
- Use variable and function names that are meaningful and **correct**
- Keep names consistent. Use either `snake_case` or `camelCase` but try not to mix both

Bad:

```r
lol <- 1:100

mydata <- data.frame(x = c("Jozef", "Gennaro", "Matt", "Morgan", "Anthony"))

f <- function(x, y, ...) {
  plot(x = x, y = y, main = "Scatter plot of x and y", ...)
}
```

Better:

```
ages <- 1:100

bioinfo_names <- data.frame(Name = c("Jozef", "Gennaro", "Matt", "Morgan", "Anthony"))

plotScatterPlot <- function(x, y, ...) {
  plot(x = x, y = y, main = "Scatter plot of x and y", ...)
}
```

Do not control program flow with comments

- this is error prone and makes it difficult or impossible to automate
- Use if/else statements instead

Bad:

```
# Download the file
#filename <- "data.tsv"
#url <- "http:://example.com/data.tsv"
#download.file(url, filename)

# Read in to a data.frame
df <- read.delim("data.tsv", sep="\t")
```

Good:

```
filename <- "data.tsv"
url <- "http:://example.com/data.tsv"

if (!file.exists(filename)) {
  download.file(url, filename)
}
df <- read.delim(filename)
```

Use a consistent style

Pick a style guide and stick with it. If using R, the styler package can automatically clean up poorly formatted code. If using Python, black is a highly opinionated formatter that is pretty popular. Although, I think ruff is currently all the rage with the Pythonistas these days.

Bad:

```
flights|>group_by(dest)|> summarize(
distance=mean( distance),speed = mean(distance/air_time, na.rm= T)) |>
ggplot(aes(x= distance, y=speed))+geom_smooth(method = "loess",span = 0.5,se = FALSE,color =
```

Good:

```
flight_plot <- flights |>
  group_by(dest) |>
  summarize(
    distance = mean(distance),
    speed = mean(distance / air_time, na.rm = TRUE)
  ) |>
  ggplot(aes(x = distance, y = speed)) +
  geom_smooth(
    method = "loess",
    span = 0.5,
    se = FALSE,
    color = "white",
    linewidth = 4
  ) +
  geom_point()
```

Don't use right hand assignment

This is R specific. I've seen this pop up with folks who are strong tidyverse adherents. I get it, that's the direction of the piping operator. However, this right-hand assignment flies in the face of basically every other programming language, and since code is primarily read rather than executed, it's much harder to scan a codebase and understand the variable assignment when the assignments can be anywhere in the pipe!

Don't do this

```
data |>
  select(...) |>
  filter(...) |>
  group_by(...) |>
  summarize(...) -> by_group
```

It's much easier to look down a script and see that `by_group` is created by all of the piped operations when assigned normally.

```
by_group <- data |>
  select(...) |>
  filter(...) |>
  group_by(...) |>
  summarize(...)
```

Summary

- Data management
  - Save all raw data and don't modify it
  - Keep good backups and make sure they work. 3-2-1 rule
  - Use consistent, meaningful filenames that make sense to computers and reflect their content or function
  - Create analysis friendly data
  - Work with/ save data in the format that it is best suited to
- Project Organization
  - Use a consistent, well-defined project structure across all projects
  - Give each project a consistent and meaningful name
  - Use the structure of the project to organize where files go
- Tracking Changes
  - Keep changes small and save changes frequently
  - If manually tracking changes do so in a logical location in a plain text document
  - Use a version control system
- Software
  - Write a short description at the top of every script about what the script does and how to use it
  - Decompose programs into functions. Don't Repeat Yourself
  - Give functions and variables meaningful names
  - Use statements for control flow instead of comments
  - Use a consistent style of coding. Use a code styler

Resources & References

Best practices

- Good Enough Practices in Scientific Computing: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510#sec004
- Backups: https://www.cisa.gov/sites/default/files/publications/data_backup_options.pdf
- File naming: https://library.si.edu/sites/default/files/filenamingorganizing_20210609.pdf
- Tidy data: https://www.jstatsoft.org/article/view/v059i10
- Frank Harrell's R Workflow for Reproducible Analysis: https://hbiostat.org/rflow/
  - FH blog is also really good for statistical concepts: https://www.fharrell.com/

Bioinformatics

- Bioinformatics Data Skills: https://www.oreilly.com/library/view/bioinformatics-data-skills/9781449367480/
- BioStars for bioinfo questions: https://www.biostars.org/
- Bioconductor common workflows: https://bioconductor.org/packages/release/BiocViews.html#___Workflow

Proficiency with computational tools

- MIT Missing Semester: https://missing.csail.mit.edu/
  - Really, check this one out

R

- R for Data Science: https://r4ds.hadley.nz/
- Advanced R: https://adv-r.hadley.nz/

- fasteR (base): https://github.com/matloff/fasteR
- Efficient R Programming: https://bookdown.org/csgillespie/efficientR/
- R performance tips: https://peerj.com/preprints/26605.pdf
- R Inferno: https://www.burns-stat.com/documents/books/the-r-inferno/
- Introduction to data science: https://rafalab.github.io/dsbook-part-1/
- Advanced data science: https://rafalab.github.io/dsbook-part-2/

Command line basics

Basic proficiency with the Unix shell is essential for anyone who wants to start doing computational work outside of their laptop and Excel. Unix shells provide an interface for interacting with Unix-like (Mac OS, Linux, etc.) operating systems and a scripting language for controlling the system. The Unix philosopy is a set of software engineering norms and concepts that guide how the tools of the Unix shell interact with one another. Learning a few of these command line tools, and how they can be strung together into what are called "pipes", is a powerful skill for developing quick and composable bioinformatics programs. Here, we'll describe some essential commands to get you started using the command line.

Accessing the terminal

First, you'll have to open the Terminal application. If you're on Mac OS, the quickest way to access your terminal is: "command + space", typing "terminal" and pressing Enter. On Windows, you'll have to install Windows Subsystem for Linux which will allow you to interact with a (default) Ubuntu OS.

Once you've opened the terminal app, you're ready to start typing commands at the command line.

Where am I?

The first command you should know is `pwd`. `pwd` will print your current working directory. This command is used to display where you are currently in the file system. For example, if I open a terminal window in my "Downloads" directory and type and hit Enter:

```
pwd
```

It will return

```
/home/gennaro/Downloads
```

indicating that I am in my "Downloads" directory.

Listing files

Now that I'm in my "Downloads" directory I want to see what files I've downloaded. To do this, I can use the `ls` command to list files in the directory.

```
ls
```

which returns:

```
BDNF-data.tsv  CORI_Candidate_SNP_draft_250528_clean.docx  differential-expression2.tsv
```

Your "Downloads" directory will of course have different files. If I need to display more information about these files, such as the time that they were created or how large they are, I can supply the `ls` command with arguments.

For example

```
ls -lah
```

Returns

```
total 15M
drwxr-xr-x  2 gennaro gennaro 4.0K Jun  1 14:52 .
drwxr-x--- 51 gennaro gennaro 4.0K Jun  1 09:34 ..
-rw-rw-r--  1 gennaro gennaro 6.8K May 30 18:00 BDNF-data.tsv
-rw-rw-r--  1 gennaro gennaro 973K May 30 17:34 CORI_Candidate_SNP_draft_250528_clean.docx
-rw-rw-r--  1 gennaro gennaro  14M May 30 12:54 differential-expression2.tsv
```

Which provides information about the file permissions, the file sizes, and when the files were created.

Learning more about a command

To learn more about what arguments are available to any of the command line programs you run, you can use the `man`, or manual, command. This command will open the user manual for the given command.

Try typing

```
man ls
```

to view all of the options available when listing files with `ls`.

Moving around

Let's say I want to move from my "Downloads" directory to my "Documents" directory. The command I have to use is `cd`, short for "change directory". We can use the `cd` command with the argument for the target directory we want to go to. For example, to move to my "Documents" directory

```
cd /home/gennaro/Documents
```

Directory shortcuts

The shell has a few shortcuts that make moving around a little easier. Running `cd` without any arguments will bring you back into your home directory.

```
cd
```

In Bash, there is an additional shortcut to specify the "/home" as well. You can use `~` in place of "/home". For example, to move into my "Documents" folder I can use

```
cd ~/Documents
```

instead of typing the full path. To go up one level in the directory you can use `..` So to go from my Documents directory 'up' into my "/home" directory I can use

```
cd ..
```

Finally, to go back to the same directory that you were just in you can use

```
cd -
```

Creating files

You can create files with the `touch` command. For example, to create an empty file in my "Downloads" directory called "A.txt" I can run

```
touch ~/Downloads/A.txt
```

Redirection

I'll add some content to this file using the `echo` command. `echo` simply prints it's arguments back out to the terminal. I'll also use what is called redirection to append the results of the `echo` command into the text file.

Redirection is a core concept in Unix pipes. It allows you to take the output from one program and use it as input to another program. In this example, I'll take the output from `echo` and redirect it to the file "A.txt" that we just created.

```
echo "This is a new line in the file" >> ~/Downloads/A.txt
echo "Here is another new line in the file" >> ~/Downloads/A.txt
```

The `>>` took the output of the `echo` command and inserted it as a new line in "A.txt". Importantly, `>>` **appended** these lines into "A.txt". If I were instead to use `>` like

```
echo "This will replace the current contents of A.txt" > ~/Downloads/A.txt
```

"A.txt" will be overwritten with the new contents. The final essential redirection operator is the pipe `|`. The pipe lets you take the output from one program and use it as input to another. I'll show an example of this later.

Displaying the content of files

The simplest way to display the contents of a file on the command line is by using the `cat` command. The `cat` command is actually designed to concatenate file together, but running it on a single file will print the entire contents of the file to the command line. For example, to print the contents of "A.txt"

```
cat ~/Downloads/A.txt
```

Will print

```
This will replace the current contents of A.txt
```

to the console. If you have a lot of text that you would like to display `cat` can result in too much information being displayed on the screen. Instead, you can use the `less` command. `less` will print the contents of the file as pages on the screen. You can use the `d` key to scroll down a page, or the `u` key to scroll up a page.

Another way to display only some of the contents of a file is to use the `head` or `tail` commands. `head -n10` will print the first 10 lines of a file, whereas `tail -n10` can be used to print the last 10 lines of a file.

Copying files

You can copy a file using the `cp` command. For example, to copy the "A.txt" file into a new file "B.txt" I can use

```
cp ~/Downloads/A.txt ~/Downloads/B.txt
```

To copy an entire directory you need to supply the `-r`, or recursive, argument to the `cp` command. For example, to create a copy of my Downloads directory inside of my Documents directory

```
cp -r ~/Downloads ~/Documents/Downloads-copy
```

Moving and renaming files

The `mv` command can be used to move files and rename them. For example, to move the "A.txt" file into my Documents directory I can use

```
mv A.txt ~/Documents
```

If I now want to change the name of that file I can also use the `mv` command. Now you need to specify the new file name instead of the location to move the file to

```
mv ~/Documents/A.txt ~/Documents/C.txt
```

Making new directories

To make a new directory you can use the `mkdir` command. To make a new directory inside of my Downloads directory I can use

```
mkdir ~/Downloads/textfiles
```

By default, the `mkdir` command doesn't allow you to create nested directories. To enable this, set the `mkdir -p` flag. For example I can create a parent folder and subfolders using

```
mkdir -p ~/Downloads/imagefiles/jpegs
```

Shell expansion

Another useful trick is to learn shell expansion. Shell expansion 'expands' the arguments. Shell expansion can be a shortcut when creating new project directories. For example

```
mkdir -p data doc scripts results/{figures,data-files,rds-files}
```

The `results/{figures,data-files,rds-files}` expands this command into

```
mkdir -p data doc scripts results/figures results/data-files results/rds-files
```

Which saves some typing. Shell expansion can also be used in other contexts. For example, I can create 260 empty text files using the following command

```
touch ~/Downloads/textfiles/{A..Z}{1..10}.txt
```

Another useful shell expansion is `*`. For example, if I needed to display the contents of each of the file we just created I could run

```
cat ~/Downloads/textfiles/*.txt
```

Removing files

Removing files on the command line can be done with the `rm` command. Unlike when using a GUI, when you remove files on the command line you cannot get them back so use `rm` wisely. To remove one of the empty files I just created I can use

```
rm ~/Downloads/textfiles/A1.txt
```

If I want to remove the entire "textfiles" directory I can use the `-r`, or recursive flag with `rm`.

```
rm -r ~/Downloads/textfiles
```

Be careful when using `rm`. A simple space can mean removing entire file systems by mistake!

Finding files

One incredibly useful but often overlooked command line tools is `find`. `find` does exactly what you expect it to do, it finds files and folders. `find` has many arguments but the simplest usage is for finding files using a specific pattern. For example, to find all text (.txt) files in a particular directory and all of its subdirectories you can use:

```
find . -name "*.txt" -type f
```

This command says, "find any file (-type f) that has a name like '.txt' ". `find` is especially powerful when combined with the `-exec` argument. For example, to remove all .txt file in a directory you can use:

```
find . -name "*.txt" -type f -exec rm {} \;
```

Downloading files

`curl` and `wget` are both command line utilities for downloading files from remote resources. `curl` will download and stream the results to your terminal by default. `wget` will save the result to a file by default.

```
curl https://www.gutenberg.org/cache/epub/100/pg100.txt
wget https://www.gutenberg.org/cache/epub/100/pg100.txt
```

Searching the contents of files

`grep` is a tool that's use to search the contents of files for specific text patterns. For example, if you wanted to find every line in a text file that contains the word "the" you could use:

```
grep "the" pg100.txt
```

`grep` also has many useful arguments. One of the most useful is that `grep` can return the count of the number of lines that are returned. For example, to count the number of lines in a text file that contain the word "the":

```
grep -c "the" pg100.txt
```

Replacing file contents

```
sed 's/find/replace/' myfile.txt
```

Piping commands

The Unix pipe is what makes the command line so powerful. You can string together small programs to build up solutions to complex problems. The pipe allows you to take the output from one program and use it as input to another program directly.

For example, suppose we wanted count the top 10 most frequently used words across all of the works of Shakespeare

```
curl https://www.gutenberg.org/cache/epub/100/pg100.txt | \
sed 's/[^a-zA-Z ]/ /g' | \
tr 'A-Z ' 'a-z\n' | \
grep '[a-z]' | \
sort | \
uniq -c | \
sort -nr -k1 | \
head -n10
```

- `curl` downloads the text file from Project Gutenberg and streams it to stdout
- `sed` replaces all characters that are not spaces or letters, with spaces.
- `tr` changes all of the uppercase letters into lowercase and converts the spaces in the lines of text to newlines (each 'word' is now on a separate line)
- `grep` includes only lines that contain at least one lowercase alphabetical character (removing any blank lines)
- `sort` sorts the list of 'words' into alphabetical order
- `uniq` counts the occurrences of each word
- `sort` sorts the occurrences numerically in descending order
- `head` shows the top 10 lines

Compressing and uncompressing

Bioinformatics and command line tools can generally work with compressed data. Data compression saves space which can be really beneficial when transferring files over the internet. `gzip` is an old but commonly used compression utility that is compatible with many command line utilities. To compress a file for example.

```
gzip pg100.txt
```

Will produce a compressed version of the "pg100.txt" file called "pg100.txt.gz". Many Unix tools can work directly with gzipped files. For example,

```
zcat pg100.txt.gz
```

Will unzip and print the contents of the file to the terminal and `zgrep` can be used to directly search the contents of a gzipped file without the need to decompress the entire file

```
zgrep -c "the" pg100.txt.gz
```

To decompress a file you use the 'un' version of the compression command, `gunzip`.

```
gunzip somefile.txt.gz
```

For-loops

The command line is also a scripting language and like any scripting language, it provides some basic control flow utilities. One of the more useful of these is the basic for loop. In Bash, the for-loop takes the form of a for each loop. the looping variable can be referred to in the loop by using the `$` syntax. For example, to loop through

```
for F in *.txt; do sort $F | uniq -c | sort -nr -k1 | head -n1 >> top-words.txt; done
```

GNU parallel

GNU parallel is a command line tool that takes away the need to use for-loops entirely. `parallel` is extremely powerful and feature filled. Importantly, it lets you run commands across multiple jobs. For example, instead of writing a for loop we can process the text files above using 8 jobs at once with `parallel`

```
parallel --jobs 8 "sort {} | uniq -c | sort -nr -k1 | head -n1 >> top-words.txt" ::: *.txt
```

Editors

You'll eventually need to edit some code or files from the terminal. Two options for code editing from the command line are `vim` and `nano`. `vim` can be more difficult to use for a beginner but is very powerful.

```
vim
```

Once you're in `vim` you can use the `i` key to enter "input" mode. "input" mode let's you type in new characters. Once you've typed away, save your work and exit with `esc + :wq`. `vim` can be difficult to get used to which lead to the most famous of StackOverflow questions: `nano` provides a more user friendly interface.

```
nano
```

Resources

- Terminus is a fun game designed to get you comfortable navigating the command line
- OverTheWire is another game designed to tech you command line tools through the lense of a 'hacker'
- vimtutor can be used to learn `vim`

R programming basics

Introduction to programming in R

There are tons of great resources for learning R. R for Data Science is probably the most popular resource for new useRs to get up to speed with slicing and dicing data in R. The *R for Data Science* book, however, is taught from the perspective of the Tidyverse. The *Tidyverse* is an opinionated set of packages and functions that help users perform data manipulations primarily on data.frames. While these packages and functions can be great for experienced users by providing ergonomic and consistent interfaces for data.frame manipulation, it is my personal belief that new users should first learn the base language, especially if their goal is to perform bioinformatics analysis.

Bioinformatics tools rely heavily on subsetting and matrix manipulations. In my experience, users who start learning R using only function from the *Tidyverse* have a difficult time understanding matrix manipulations and subsetting operations common in bioinformatics workflows. This becomes especially important when using SummarizedExperiments - the backbone of many bioinformatics data structures in R.

For this reason, we're going to focus on learning R from the ground up using functions that exist primarily in the base language. A great resource for learning base R quickly is Norm Matloff's *fasteR* which can be found here.

Subsetting in R

You may have only ever encountered R from the perspective of the tidyverse. `tidyverse` functions provide useful abstractions for munging tidy data however, most genomics data is often best represented and operated on as matrices. Keeping your data in matrix format can provide many benefits as far as speed and code clarity, which in turn helps to ensure correctness. You can think of matrices as just fancy 2D versions of vectors. So what are vectors?

Vectors are the main building blocks of most R analyses. Whenever you use the `c()` function ('concatenate'), like: `x <- c('a', 'b', 'c')` you're creating a vector. Vectors hold R objects and are the building block of more complex structures in R.

NOTE: the following is heavily inspired by Norm Matloff's excellent fasteR tutorial. Take a look there to get a brief and concise overview base R. You should also check out the first few

chapters of Hadley Wickham's amazing book Advanced R. The first edition contains some more information on base R.

Subsetting vectors

Below, we'll use the built-in R constant called `LETTERS`. The `LETTERS` vector is simply a 'list' of all uppercase letters in the Roman alphabet.

```
LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

We can subset the vector by position. For example, to get the 3rd letter we use the `[` operator and the position we want to extract.

```
LETTERS[3]
```

```
[1] "C"
```

We can also use a range of positions. The notation `3:7` is a shortcut that generates the numbers, 3, 4, 5, 6, 7.

```
LETTERS[3:7]
```

```
[1] "C" "D" "E" "F" "G"
```

We don't have to select sequential elements either. We can extract elements by using another vector of positions.

```
LETTERS[c(7, 5, 14, 14, 1, 18, 15)]
```

```
[1] "G" "E" "N" "N" "A" "R" "O"
```

Vectors become really powerful when we start combining them with logical operations. R supports all of the usual logical and comparison operators you can expect from a programming language, `<`, `>`, `==`, `!=`, `<=`, `>=`, `%in%`, `&` and `|`.

```r
my_favorite_letters <- c("A", "B", "C")

# See that this produces a logical vector of (TRUE/FALSE) values
# TRUE when LETTERS is one of my_favorite_letters and FALSE otherwise
LETTERS %in% my_favorite_letters
```

```
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
```

```r
# We can use that same expression to filter the vector
LETTERS[LETTERS %in% my_favorite_letters]
```

```
[1] "A" "B" "C"
```

This same kind of subsetting works on vectors that contain numeric data as well. For example, we can filter the measurements of annual flow of water through the Nile river like so:

Nile is another built-in dataset

```r
# Any values strictly greater than 1200
Nile[Nile > 1200]
```

```
[1] 1210 1230 1370 1210 1250 1260 1220
```

```r
# Any even number - `%%` is the modulus operator
Nile[Nile %% 2 == 0]
```

```
 [1] 1120 1160 1210 1160 1160 1230 1370 1140 1110  994 1020  960 1180  958 1140
[16] 1100 1210 1150 1250 1260 1220 1030 1100  774  840  874  694  940  916  692
[31] 1020 1050  726  456  824  702 1120 1100  832  764  768  864  862  698  744
[46]  796 1040  944  984  822 1010  676  846  812  742 1040  860  874  848  890
[61]  744  838 1050  918  986 1020  906 1170  912  746  718  714  740
```

At this point it's important to take a step back and appreciate what R is doing. Each of the comparison operators that we used above is *vectorized*. This means that the comparison is applied to all elements of the vector at one time. If you're used to a programming language like Python this might seem foreign at first. In Python, you would have to write a list comprehension to filter observations from a list that meet a certain condition. For example, `[x for x in Nile if x > 1200]`. However in R, most functions and operators are *vectorized* allowing us to do things like `Nile > 1200` and have the comparison applied to all of the elements of the vector automatically.

Subsetting data.frames

But these are just one dimensional vectors. In R, we usually deal with data.frames (tibbles for you tidyverse folks) and matrices. Lucky for us, the subsetting operations we learned for vectors work the same way for data.frames and matrices.

Let's take a look at the built-in `ToothGrowth` dataset. The data consists of the length of odontoblasts in 60 guinea pigs receiving one of three levels of vitamin C by one of two delivery methods.

```
head(ToothGrowth)
```

```
   len supp dose
1  4.2   VC  0.5
2 11.5   VC  0.5
3  7.3   VC  0.5
4  5.8   VC  0.5
5  6.4   VC  0.5
6 10.0   VC  0.5
```

The dollar sign `$` is used to extract an individual column from the data.frame, which is just a vector.

```
head(ToothGrowth$len)
```

```
[1]  4.2 11.5  7.3  5.8  6.4 10.0
```

We can also use the `[[` to get the same thing. Double-brackets come in handy when your columns are not valid R names since `$` only works when columns are valid names.

```
head(ToothGrowth[["len"]])
```

```
[1]  4.2 11.5  7.3  5.8  6.4 10.0
```

When subsetting a data.frame in base R, the general scheme is:

```
df[the rows you want, the columns you want]
```

So in order to get the 5th row of the first column we could do:

```
ToothGrowth[5, 1]
```

```
[1] 6.4
```

Again, we can combine this kind of thinking to extract rows and columns matching logical conditions. For example, if we want to get all of the animals administered orange juice ('OJ')

```
ToothGrowth[ToothGrowth$supp == "OJ", ]
```

```
    len supp dose
31 15.2   OJ  0.5
32 21.5   OJ  0.5
33 17.6   OJ  0.5
34  9.7   OJ  0.5
35 14.5   OJ  0.5
36 10.0   OJ  0.5
37  8.2   OJ  0.5
38  9.4   OJ  0.5
39 16.5   OJ  0.5
40  9.7   OJ  0.5
41 19.7   OJ  1.0
42 23.3   OJ  1.0
43 23.6   OJ  1.0
44 26.4   OJ  1.0
45 20.0   OJ  1.0
46 25.2   OJ  1.0
47 25.8   OJ  1.0
48 21.2   OJ  1.0
49 14.5   OJ  1.0
50 27.3   OJ  1.0
51 25.5   OJ  2.0
52 26.4   OJ  2.0
53 22.4   OJ  2.0
54 24.5   OJ  2.0
55 24.8   OJ  2.0
56 30.9   OJ  2.0
57 26.4   OJ  2.0
58 27.3   OJ  2.0
59 29.4   OJ  2.0
60 23.0   OJ  2.0
```

We can also combine logical statements. For example, to get all of the rows for animals administered orange juice and with odontoblast length ('len') less than 10.

```
ToothGrowth[ToothGrowth$supp == "OJ" & ToothGrowth$len < 10, ]
```

```
   len supp dose
34 9.7   OJ  0.5
37 8.2   OJ  0.5
38 9.4   OJ  0.5
40 9.7   OJ  0.5
```

```
# We can also use the bracket notation to select rows and columns at the same time
# Although this gets a little difficult to read
ToothGrowth[ToothGrowth$supp == "OJ" & ToothGrowth$len < 10, c("len", "supp")]
```

```
   len supp
34 9.7   OJ
37 8.2   OJ
38 9.4   OJ
40 9.7   OJ
```

It gets annoying typing `ToothGrowth` every time we want to subset the data.frame. Base R has a very useful function called `subset()` that can help us type less. `subset()` essentially 'looks inside' the data.frame for the given columns and evaluates the expression without having to explicitly tell R where to find the columns. Think of it like `dplyr::filter()`, if you are familiar with that function.

```
subset(ToothGrowth, supp == "OJ" & len < 10)
```

```
   len supp dose
34 9.7   OJ  0.5
37 8.2   OJ  0.5
38 9.4   OJ  0.5
40 9.7   OJ  0.5
```

Subsetting Lists

Another data structure to be aware of, which is used frequently, is the `List`. We've actually already encountered Lists above. data.frames are really just Lists where each vector contains the same data type and all List elements are the same length.

We can create a List in R using the `list()` function. Notice how each list element has a name and can contain a different type of data and number of data elements

```
l <- list(
  element1 = c(1, 10, 12, 3, 6, 12, 13, 2, 5, 6, 3, 7),
  element2 = c("a", "b", "c"),
  element3 = c(TRUE, TRUE, FALSE, FALSE, FALSE),
  element4 = c(0.001, 0.05, 0.86, 1.098, 345.0)
)
```

Lists can be tricky at first. To extract the data from a particular list element you can use the `[[` or the `$` (as in the case of data.frames above). Like vectors, you can use either the index or the name of the element you wish to extract.

```
l[[1]]
```

```
 [1]  1 10 12  3  6 12 13  2  5  6  3  7
```

```
l[["element1"]]
```

```
 [1]  1 10 12  3  6 12 13  2  5  6  3  7
```

```
l$element1
```

```
 [1]  1 10 12  3  6 12 13  2  5  6  3  7
```

What is returned if you only use the single bracket `[`?

```
l[1]
```

```
$element1
 [1]  1 10 12  3  6 12 13  2  5  6  3  7
```

You get another List, but now with a single element. This behavior might seem unintuitive at first, but it can be very useful for creating new lists.

```
numeric_l <- l[c(1, 4)]
```

Subsetting matrices

Matrices behave much like data.frames but unlike data.frames matrices can only contain one type of data. This might sound like a limitation at first but you'll soon come to realize that matrices are very powerful (and fast) to work with in R.

```
set.seed(123)

# Create some random data that looks like methylation values
(m <- matrix(
  data = runif(6 * 10),
  ncol = 6,
  dimnames = list(
    paste0("CpG.", 1:10),
    paste0("Sample", 1:6)
  )
))
```

```
          Sample1    Sample2   Sample3    Sample4   Sample5    Sample6
CpG.1  0.2875775 0.95683335 0.8895393 0.96302423 0.1428000 0.04583117
CpG.2  0.7883051 0.45333416 0.6928034 0.90229905 0.4145463 0.44220007
CpG.3  0.4089769 0.67757064 0.6405068 0.69070528 0.4137243 0.79892485
CpG.4  0.8830174 0.57263340 0.9942698 0.79546742 0.3688455 0.12189926
CpG.5  0.9404673 0.10292468 0.6557058 0.02461368 0.1524447 0.56094798
CpG.6  0.0455565 0.89982497 0.7085305 0.47779597 0.1388061 0.20653139
CpG.7  0.5281055 0.24608773 0.5440660 0.75845954 0.2330341 0.12753165
CpG.8  0.8924190 0.04205953 0.5941420 0.21640794 0.4659625 0.75330786
CpG.9  0.5514350 0.32792072 0.2891597 0.31818101 0.2659726 0.89504536
CpG.10 0.4566147 0.95450365 0.1471136 0.23162579 0.8578277 0.37446278
```

If we want to extract the value for CpG.3 for Sample3

```
m[3, 3]
```

```
[1] 0.6405068
```

Or all values of CpG.3 for every sample

```
m[3, ]
```

```
  Sample1   Sample2   Sample3   Sample4   Sample5   Sample6
0.4089769 0.6775706 0.6405068 0.6907053 0.4137243 0.7989248
```

```
# Or refer to the row by it's name
m["CpG.3", ]
```

```
  Sample1   Sample2   Sample3   Sample4   Sample5   Sample6
0.4089769 0.6775706 0.6405068 0.6907053 0.4137243 0.7989248
```

Or all CpGs for Sample3

```
m[, 3]
```

```
    CpG.1     CpG.2     CpG.3     CpG.4     CpG.5     CpG.6     CpG.7     CpG.8
0.8895393 0.6928034 0.6405068 0.9942698 0.6557058 0.7085305 0.5440660 0.5941420
    CpG.9    CpG.10
0.2891597 0.1471136
```

```
# Or refer to the column by it's name
m[, "Sample3"]
```

```
    CpG.1     CpG.2     CpG.3     CpG.4     CpG.5     CpG.6     CpG.7     CpG.8
0.8895393 0.6928034 0.6405068 0.9942698 0.6557058 0.7085305 0.5440660 0.5941420
    CpG.9    CpG.10
0.2891597 0.1471136
```

We can also apply a mask to the entire matrix at once. For example, the following will mark any value that is greater than 0.5 with `TRUE`

```
m > 0.5
```

```
       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
CpG.1    FALSE    TRUE    TRUE    TRUE   FALSE   FALSE
CpG.2     TRUE   FALSE    TRUE    TRUE   FALSE   FALSE
CpG.3    FALSE    TRUE    TRUE    TRUE   FALSE    TRUE
CpG.4     TRUE    TRUE    TRUE    TRUE   FALSE   FALSE
CpG.5     TRUE   FALSE    TRUE   FALSE   FALSE    TRUE
CpG.6    FALSE    TRUE    TRUE   FALSE   FALSE   FALSE
CpG.7     TRUE   FALSE    TRUE    TRUE   FALSE   FALSE
CpG.8     TRUE   FALSE    TRUE   FALSE   FALSE    TRUE
CpG.9     TRUE   FALSE   FALSE   FALSE   FALSE    TRUE
CpG.10   FALSE    TRUE   FALSE   FALSE    TRUE   FALSE
```

We can use this kind of masking to filter rows of the matrix using some very helpful base R functions that operate on matrices. For example, to get only those CpGs where 3 or more samples have a value > 0.5 we can use the `rowSums()` like so:

```
m[rowSums(m > 0.5) > 3, ]
```

```
        Sample1   Sample2   Sample3   Sample4   Sample5   Sample6
CpG.3 0.4089769 0.6775706 0.6405068 0.6907053 0.4137243 0.7989248
CpG.4 0.8830174 0.5726334 0.9942698 0.7954674 0.3688455 0.1218993
```

This pattern is very common when dealing with sequencing data. Base R functions like `rowSums()` and `colMeans()` are specialized to operate over matrices and are the most efficient way to summarize matrix data. The R package matrixStats also contains highly optimized functions for operating on matrices.

Compare the above to the `tidy` solution given the same matrix.

```
tidyr::as_tibble(m, rownames = "CpG") |>
  tidyr::pivot_longer(!CpG, names_to = "SampleName", values_to = "beta") |>
  dplyr::group_by(CpG) |>
  dplyr::mutate(n = sum(beta > 0.5)) |>
  dplyr::filter(n > 3) |>
  tidyr::pivot_wider(id_cols = CpG, names_from = "SampleName", values_from = "beta") |>
  tibble::column_to_rownames(var = "CpG") |>
  data.matrix()
```

```
        Sample1   Sample2   Sample3   Sample4   Sample5   Sample6
CpG.3 0.4089769 0.6775706 0.6405068 0.6907053 0.4137243 0.7989248
CpG.4 0.8830174 0.5726334 0.9942698 0.7954674 0.3688455 0.1218993
```

There's probably some kind of tidy solution using `across()` that I'm missing but this is how most of the tidy code in the wild that I have seen looks

Data Wrangling

Now that we've got a handle on some different R data types, and how to slice and dice them, we can start learning basic data cleaning and exploratory data analysis. We'll focus on using data.frames since they're the primary workhorse of data analysis in R. But remember, data.frames are just lists of vectors with some special rules, so many concepts you learn will apply to data.frames but also apply to vectors and lists.

Data import

We'll use the built in `penguins_raw` dataset to learn some basic data cleaning. This dataset is built into R version 4.5 so you can just load it by running `penguins_raw` if you have that R version installed. However, to illustrate data import, we'll read in the data from an external source.

The `read.csv()` function can read in comma-separated value files that are located either on your local machine or from remote sources if provided a URL.

```
url <- "https://raw.githubusercontent.com/allisonhorst/palmerpenguins/refs/heads/main/inst/ex
penguins <- read.csv(url)
```

The `read.csv()` function has many options for reading in data. If you want to learn about all of the options any particular R function has, you can prefix the function name with a `?` like, `?read.csv()` to bring up the help documentation.

Viewing data

The code above read the data into data.frame that we called `penguins`. We can take a look at the first few rows of the `penguins` data.frame using the `head()` function.

```
head(penguins)
```

```
  studyName Sample.Number                                Species Region    Island
1   PAL0708             1 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
2   PAL0708             2 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
3   PAL0708             3 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
4   PAL0708             4 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
5   PAL0708             5 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
6   PAL0708             6 Adelie Penguin (Pygoscelis adeliae) Anvers Torgersen
               Stage Individual.ID Clutch.Completion   Date.Egg
1 Adult, 1 Egg Stage          N1A1               Yes 2007-11-11
2 Adult, 1 Egg Stage          N1A2               Yes 2007-11-11
3 Adult, 1 Egg Stage          N2A1               Yes 2007-11-16
4 Adult, 1 Egg Stage          N2A2               Yes 2007-11-16
5 Adult, 1 Egg Stage          N3A1               Yes 2007-11-16
6 Adult, 1 Egg Stage          N3A2               Yes 2007-11-16
  Culmen.Length..mm. Culmen.Depth..mm. Flipper.Length..mm. Body.Mass..g.    Sex
1               39.1              18.7                 181          3750   MALE
2               39.5              17.4                 186          3800 FEMALE
```

```
3               40.3              18.0                 195          3250 FEMALE
4                 NA                NA                  NA            NA   <NA>
5               36.7              19.3                 193          3450 FEMALE
6               39.3              20.6                 190          3650   MALE
  Delta.15.N..o.oo. Delta.13.C..o.oo.                      Comments
1                NA                NA Not enough blood for isotopes.
2           8.94956         -24.69454                          <NA>
3           8.36821         -25.33302                          <NA>
4                NA                NA          Adult not sampled.
5           8.76651         -25.32426                          <NA>
6           8.66496         -25.29805                          <NA>
```

If we want to get a general overview of the data, we can use the `str()` function.

```
str(penguins)
```

```
'data.frame':   344 obs. of  17 variables:
 $ studyName          : chr  "PAL0708" "PAL0708" "PAL0708" "PAL0708" ...
 $ Sample.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Species            : chr  "Adelie Penguin (Pygoscelis adeliae)" "Adelie Penguin (Pygoscel
 $ Region             : chr  "Anvers" "Anvers" "Anvers" "Anvers" ...
 $ Island             : chr  "Torgersen" "Torgersen" "Torgersen" "Torgersen" ...
 $ Stage              : chr  "Adult, 1 Egg Stage" "Adult, 1 Egg Stage" "Adult, 1 Egg Stage"
 $ Individual.ID      : chr  "N1A1" "N1A2" "N2A1" "N2A2" ...
 $ Clutch.Completion  : chr  "Yes" "Yes" "Yes" "Yes" ...
 $ Date.Egg           : chr  "2007-11-11" "2007-11-11" "2007-11-16" "2007-11-16" ...
 $ Culmen.Length..mm. : num  39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ Culmen.Depth..mm.  : num  18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
 $ Flipper.Length..mm.: int  181 186 195 NA 193 190 181 195 193 190 ...
 $ Body.Mass..g.      : int  3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
 $ Sex                : chr  "MALE" "FEMALE" "FEMALE" NA ...
 $ Delta.15.N..o.oo.  : num  NA 8.95 8.37 NA 8.77 ...
 $ Delta.13.C..o.oo.  : num  NA -24.7 -25.3 NA -25.3 ...
 $ Comments           : chr  "Not enough blood for isotopes." NA NA "Adult not sampled." ...
```

There are a few external packages that are also very useful for getting summaries of data.frames. `Hmsic::describe()` and `skimr::skim()` are two standouts.

One of the most basic ways to get an idea of the data is to summarize each variable. There are a few functions we can use to get summaries of the data. The `table()` function will count the number of occurrences of each type in a vector.

For example, how many observations of each species of penguin are in the dataset?

```
table(penguins$Species)
```

```
      Adelie Penguin (Pygoscelis adeliae)
                                      152
Chinstrap penguin (Pygoscelis antarctica)
                                       68
        Gentoo penguin (Pygoscelis papua)
                                      124
```

R also provides the typical summary functions that you would expect from a statistical programming language such as `mean()`, `median()`, `min()`, and `max()`, and `length()`.

For example, what is the mean flipper length?

```
mean(penguins$Flipper.Length..mm.)
```

```
[1] NA
```

On no! This returned NA but there is clearly data in this column. What happened? Missing data is commonly observed across all data domains. NAs simply represent unknown values in this context and it's impossible to know how to take the mean of a value that's known with a value that's unknown. It is for this reason that many R functions have an argument called `na.rm=`. Setting `na.rm=TRUE` in these functions tells R to ignore the NA values.

```
mean(penguins$Flipper.Length..mm., na.rm = TRUE)
```

```
[1] 200.9152
```

Now we can see that the mean flipper length is ~200 mm across all observations in the dataset. Another useful function is `summary()`. Running `summary()` on a numeric vector returns a lot of useful information.

```
summary(penguins$Flipper.Length..mm., na.rm = TRUE)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  172.0   190.0   197.0   200.9   213.0   231.0       2
```

Finally, if you are using and IDE like Rstudio or Positron you can run the `View()` function on your data.frame. This will bring up an interactive data viewer.

Splitting data

You may have noticed above that we computed the mean flipper length across all species of penguins. But do all species have the same mean? A common pattern in R is called "split-apply-combine". This pattern means, split the data into groups you're interested in, apply a function to each of those groups, and combine the results. One such function that performs this operation is called `tapply()`. `tapply()` will split the data by a given variable into groups and apply a function to each group.

For example, to find the mean flipper length for each species we could use

```
tapply(penguins$Flipper.Length..mm., penguins$Species, mean, na.rm = TRUE)
```

```
      Adelie Penguin (Pygoscelis adeliae)
                                 189.9536
Chinstrap penguin (Pygoscelis antarctica)
                                 195.8235
        Gentoo penguin (Pygoscelis papua)
                                 217.1870
```

The basic format of the `tapply()` function is

```
tapply("data to split", "what to split by", "what to compute")
```

Splitting can also by applied directly to data.frames using the `split()` function. Let's say we wanted to split the penguins data.frame into one data.frame for each species. We can use the `split()` function for this purpose.

```
by_species <- split(penguins, f = penguins$Species)
```

This function returns a list of data.frames, one for each species in the original data.frame. Use `names(by_species)` to see what each list element is named. To extract the first data.frame from this list, which contains Adelie penguin data only, we can subset the list of data.frames.

```
adelie <- by_species[[1]]
```

Performing operations on lists is such a common task in R that a function exists specifically to apply functions to list elements. This function is called `lapply()`. `lapply()` takes a list and a function and applies that function to each list element. The `lapply()` function returns the results as a list. We'll learn more about this later in the functional programming section.

For example, to see how many rows are in each of the data.frames in the "by_species" list:

```r
lapply(by_species, nrow)
```

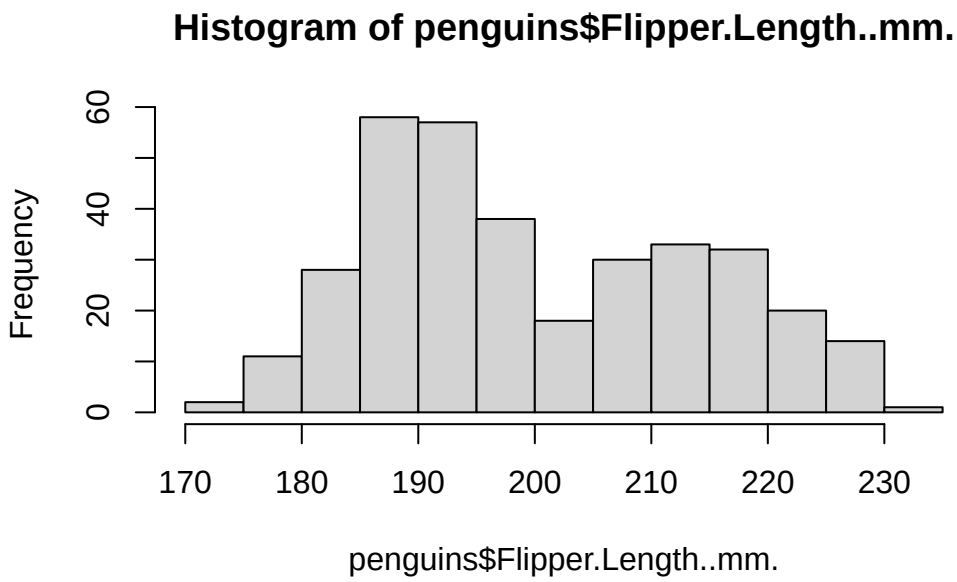
```
$`Adelie Penguin (Pygoscelis adeliae)`
[1] 152

$`Chinstrap penguin (Pygoscelis antarctica)`
[1] 68

$`Gentoo penguin (Pygoscelis papua)`
[1] 124
```

Plotting data

Data cleaning and data visualization go hand-in-hand. To effectively clean data, you should be examining the changes you're making in real time. Base R actually has very powerful graphics capabilities for quickly visualizing data. Packages like ggplot2 and lattice provide powerful alternatives to base R plots. We'll cover `ggplot2` later. For now, base R plotting can provide all we need for exploratory analyses.

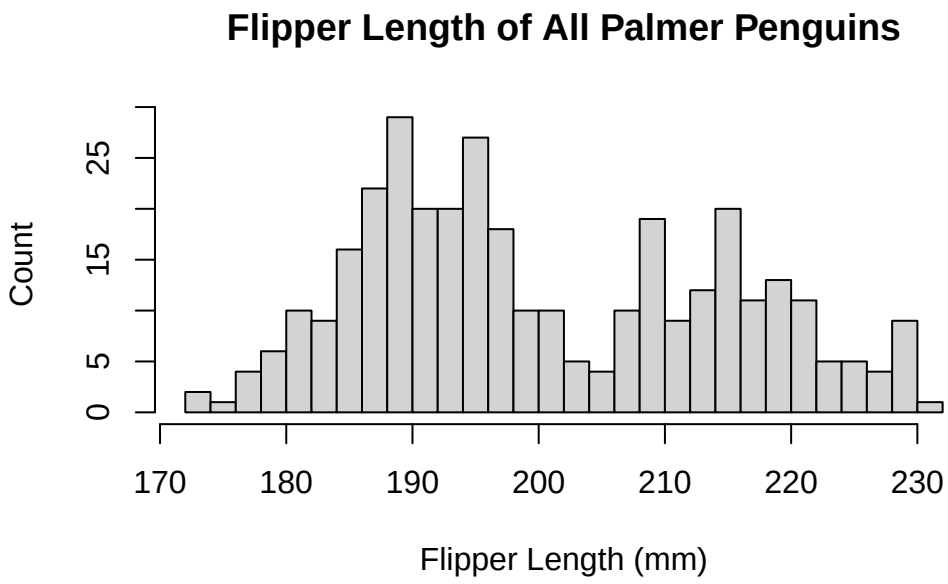
One of the most useful plots for numeric data is a histogram. Histograms bin the data and plot how many occurrences of a particular bin are present. This plot allows you to get an idea of the numeric summary of a variable. To plot a histogram of a numeric variable we can use the `hist()` function.

```r
hist(penguins$Flipper.Length..mm.)
```



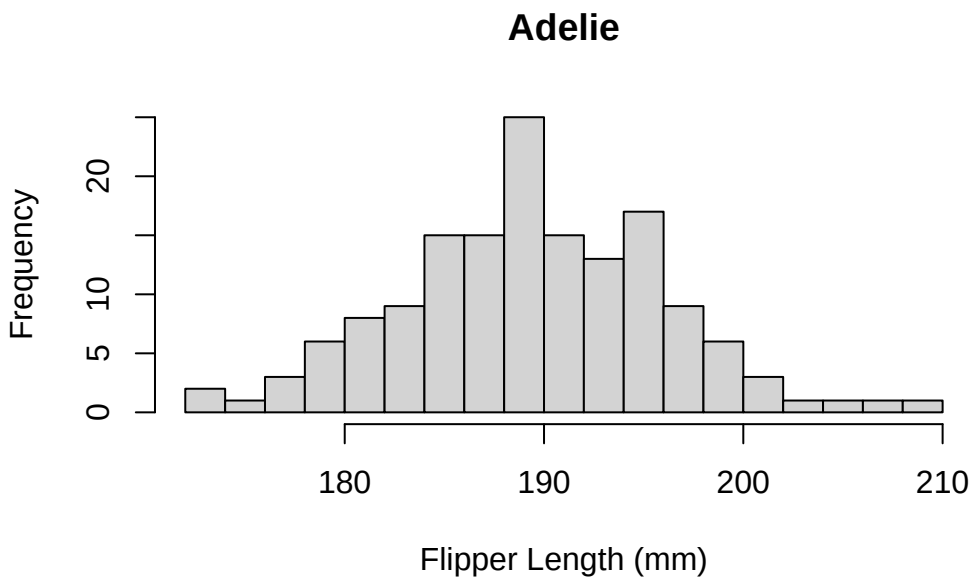
The histogram has a few arguments we can use to adjust the plot. Use `?hist()` to see the full list. Below, we can adjust the axes to be more informative and modify the number of bins we're computing.

```
hist(penguins$Flipper.Length..mm.,
     breaks = 30,
     main = "Flipper Length of All Palmer Penguins",
     xlab = "Flipper Length (mm)",
     ylab = "Count"
     )
```

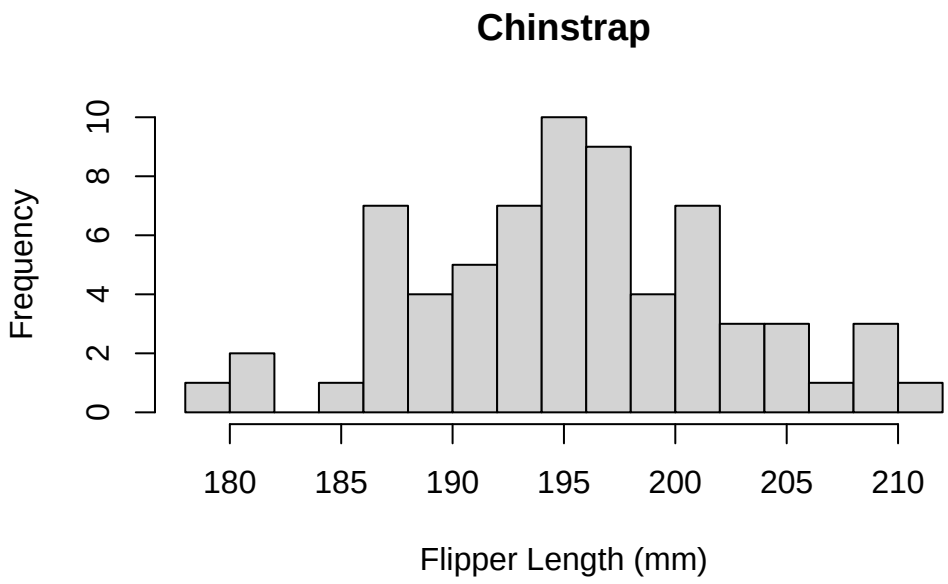


The distribution appears to be bimodal. Is this because there are different species present in this plot? We can check by applying the `hist()` function to each of the groups using the list of data.frames from above.

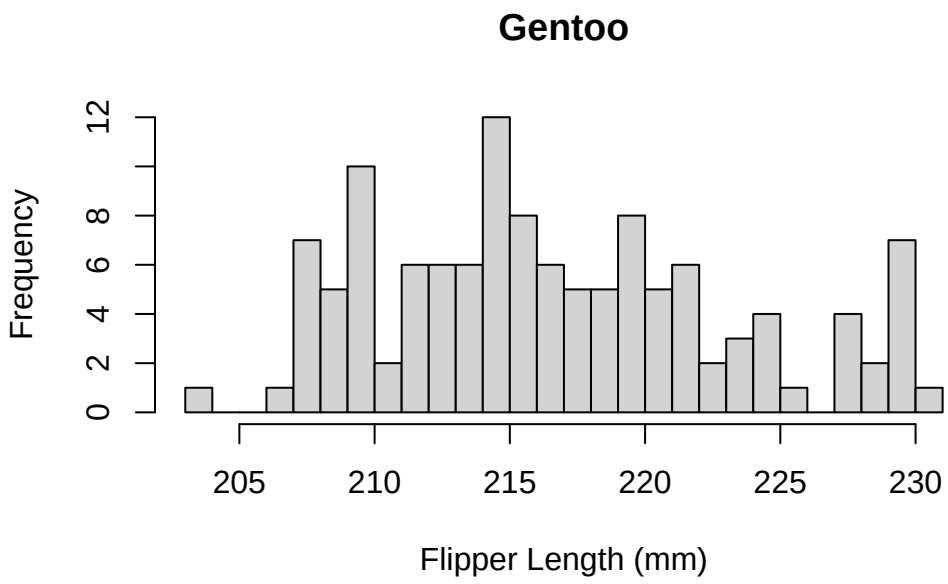
```
# Adelie penguins
hist(by_species[[1]]$Flipper.Length..mm.,
     breaks = 20,
     main = "Adelie",
     xlab = "Flipper Length (mm)")
```




```
# Chinstrap penguins
hist(by_species[[2]]$Flipper.Length..mm.,
     breaks = 20,
     main = "Chinstrap",
     xlab = "Flipper Length (mm)")
```

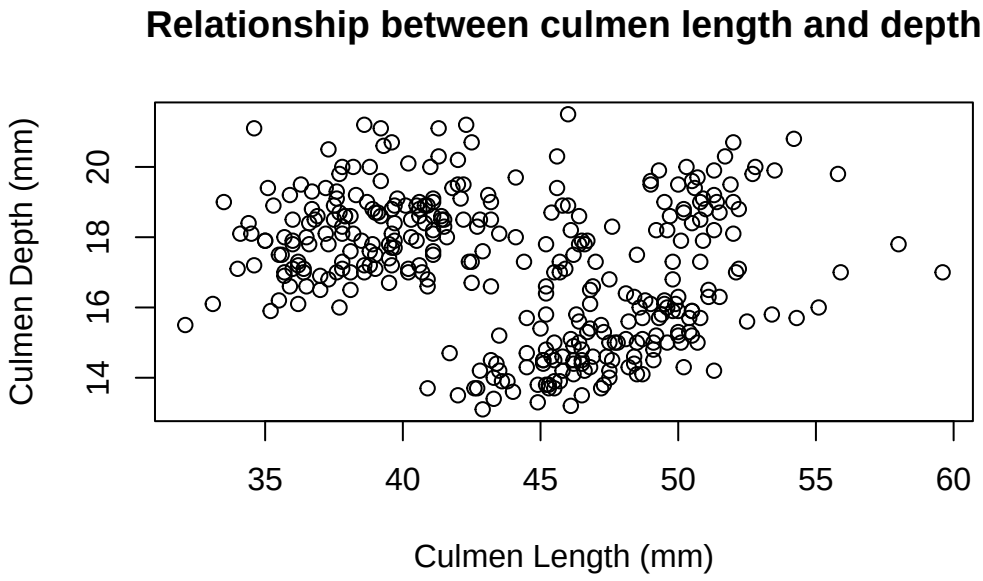



```
# Gentoo penguins
hist(by_species[[3]]$Flipper.Length..mm.,
     breaks = 20,
     main = "Gentoo",
     xlab = "Flipper Length (mm)")
```



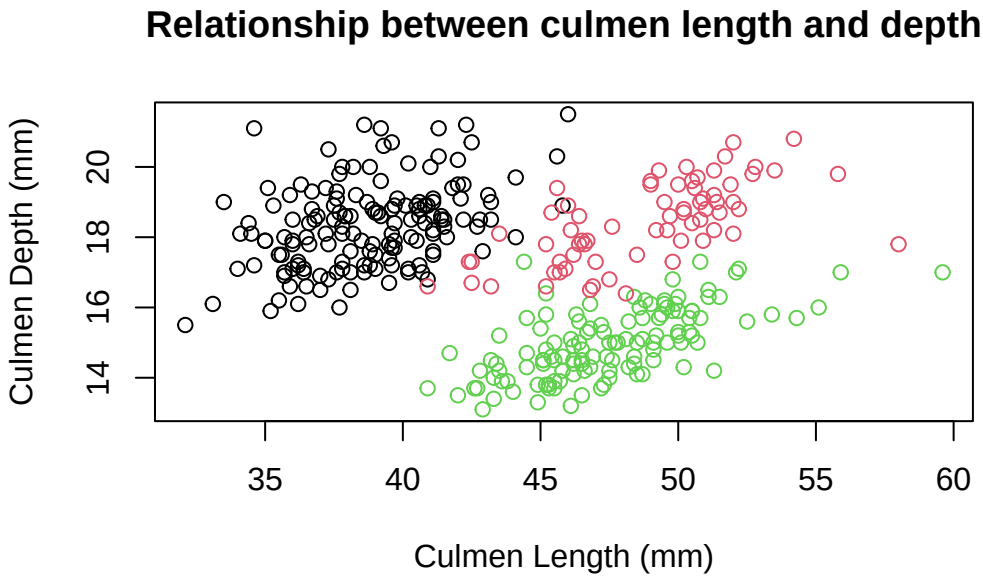
Histograms are useful for plotting the distribution of a single numeric variable. Often, we wish to see how two variables are related. The `plot()` function provides a way to create a scatter plot of two variables against each other on the same plot. For example, to plot the culmen length vs the culmen depth:

```
plot(x = penguins$Culmen.Length..mm.,
     y = penguins$Culmen.Depth..mm.,
     xlab = "Culmen Length (mm)",
     ylab = "Culmen Depth (mm)",
     main = "Relationship between culmen length and depth"
     )
```



Again, these data points seem to be split by the different species of penguins present in the dataset. We can color these points using an additional variable in the call to the `plot()` function.

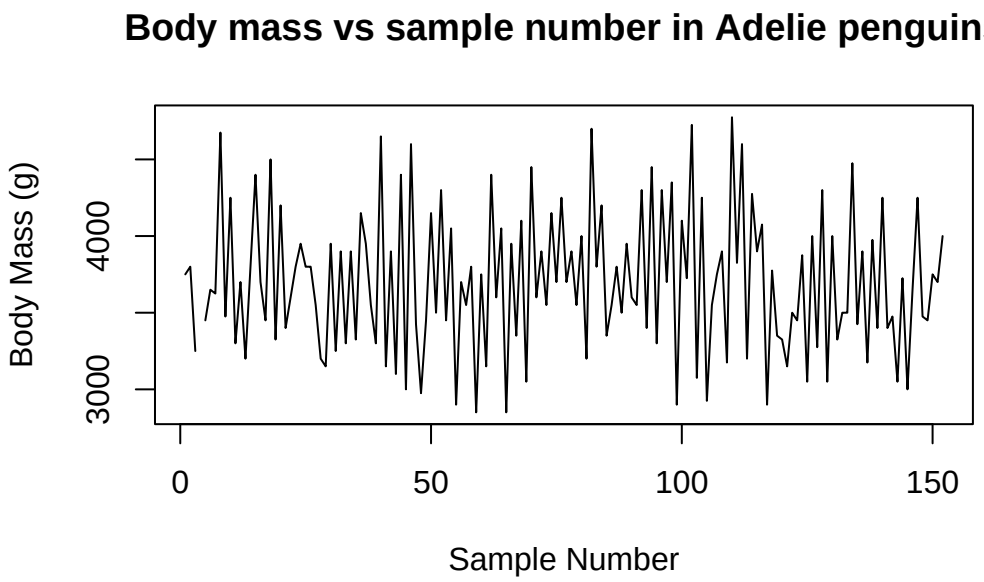
```
plot(x = penguins$Culmen.Length..mm.,
     y = penguins$Culmen.Depth..mm.,
     col = factor(penguins$Species),
     xlab = "Culmen Length (mm)",
     ylab = "Culmen Depth (mm)",
     main = "Relationship between culmen length and depth"
     )
```



You may have noticed that we did something special to that vector of species. We wrapped it in the `factor()` function. In R, factors are used when we have categorical values. R uses factors to represent the different levels of each category. It may not seem important now, but factors are very useful for statistical analysis. We'll build on this topic shortly.

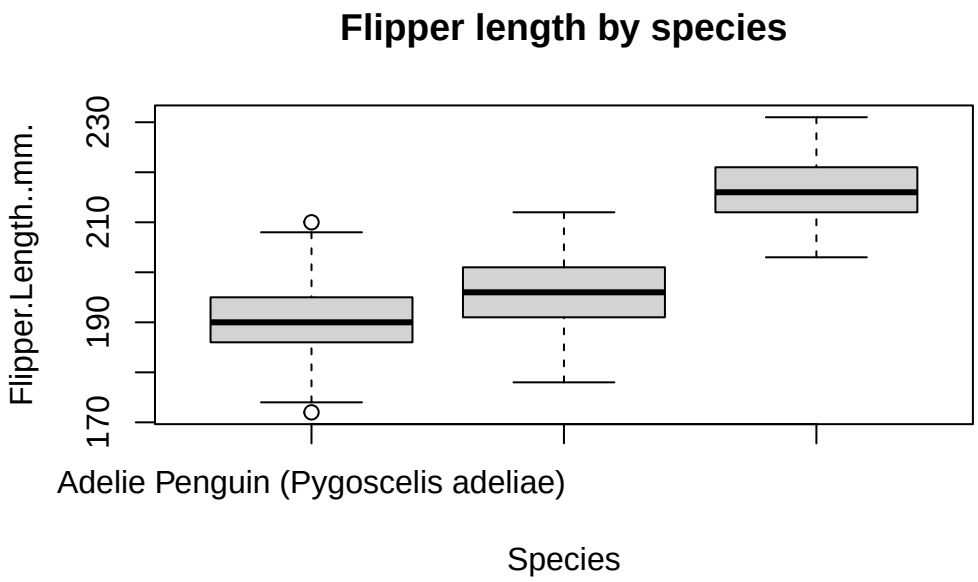
The `plot()` function is very versatile. You can use it to create line plots as well, to show trends of variables over time. Here is a contrived example of using `plot()` to create a simple line chart.

```
plot(
  x = adelie$Sample.Number,
  y = adelie$Body.Mass..g.,
  type = "l",
  main = "Body mass vs sample number in Adelie penguins",
  xlab = "Sample Number",
  ylab = "Body Mass (g)"
  )
```



Scatter plots are useful for displaying two numeric values against each other. However you'll often want to compare numeric values across categorical values. One such plot for doing this is called a boxplot. Boxplots show the median and interquartile (IQR) range for a set of data. The 'whiskers' of the boxplot display the 1.5 x IQR of the data. We can plot a boxplot of the flipper length for each species using the `boxplot()` function.

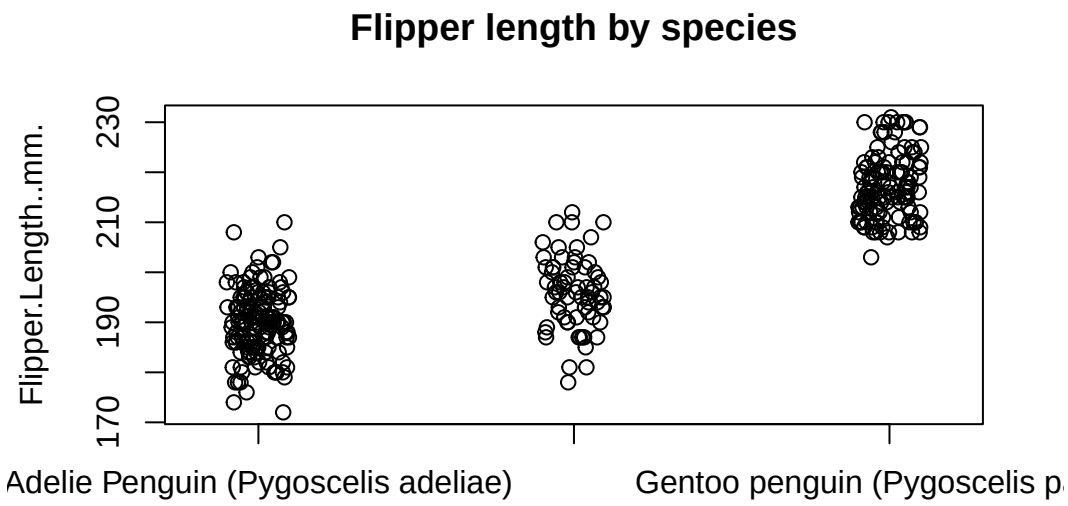
```
boxplot(
  Flipper.Length..mm. ~ Species,
  data = penguins,
  main = "Flipper length by species"
  )
```



The `boxplot()` function is a little funny because it uses formula notation. Many functions in R accept formula notation as input. In this case, the formula notation means "plot the flipper length as a function of the species type". You may have also noticed that we gave this function the penguins data.frame as the value for the `data=` argument. This allowed the boxplot function to use the column names without us having to explicitly say that they came from the penguins data.frame.

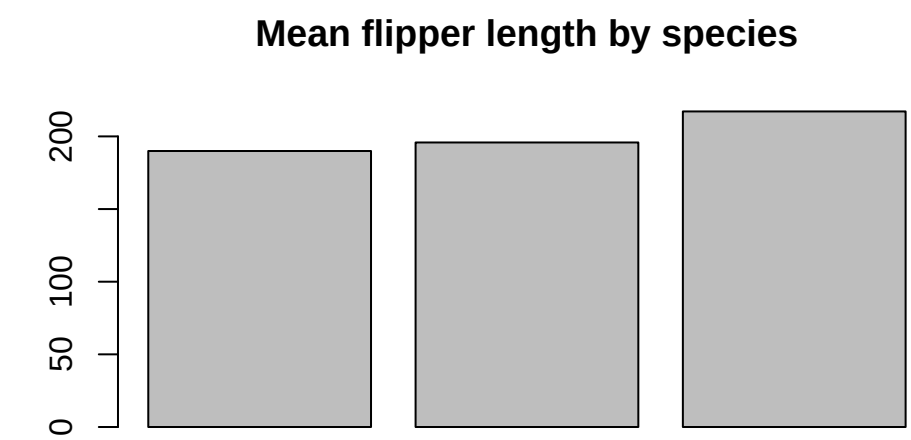
If there's not too much data, boxplots can actually hide a lot of information. In this case, another option is to use a stripchart. A stripchart shows every data point on the plot instead of a depiction of the distribution as in the boxplot.

```
stripchart(
  Flipper.Length..mm. ~ Species,
  data = penguins,
  method = "jitter",
  pch = 1,
  main = "Flipper length by species",
  vertical = TRUE
  )
```



Bar charts are another common way that people show summaries of data. To display a barplot of data, we can use the `barplot()` function. To create a barplot of the mean flipper length for each species we first need to summarize the flipper length for each group and then supply these summaries to the `barplot()` function.

```
species_means <- tapply(penguins$Flipper.Length..mm., penguins$Species, mean, na.rm = TRUE)
barplot(species_means, main = "Mean flipper length by species")
```



There are many ways to modify these default plot types and make the charts look great. Take the time to look into some of base R graphing capabilities. A really good tutorial for learning about base R plotting can be found here. Also take a look at `?pairs()`, `?dotchart()`, and `?coplot()` for some other useful base R graphics functions for creating quick plots.

Filtering data

We've already learned how to subset a data.frame but let's just take a step back and quickly revisit subsetting data.frames. To subset a data.frame means to select rows of a data.frame based on some condition that you're interested in. This subsetting can be accomplished using the `[` operator and setting conditions by specifying `df[the rows we want, the cols we want]` syntax.

For example, in the above plots it looked like the Gentoo penguins typically had the largest flippers. It looked like a value > 205 roughly separated the Gentoo penguins from the others. How could we select the rows where the flipper length is greater than or equal to 205 and count the species types?

```
big_flippers <- penguins[penguins$Flipper.Length..mm. >= 205, ]
table(big_flippers$Species)
```

```
      Adelie Penguin (Pygoscelis adeliae)
                                        3
Chinstrap penguin (Pygoscelis antarctica)
                                        8
        Gentoo penguin (Pygoscelis papua)
                                      122
```

We can combine conditions as well to select on multiple conditions. For example, let's select the female penguins with flippers >= 205 mm.

```
head(penguins[penguins$Sex == "FEMALE" & penguins$Flipper.Length..mm. >= 205, ])
```

```
    studyName Sample.Number                           Species Region Island
NA       <NA>            NA                              <NA>   <NA>   <NA>
153   PAL0708             1 Gentoo penguin (Pygoscelis papua) Anvers Biscoe
155   PAL0708             3 Gentoo penguin (Pygoscelis papua) Anvers Biscoe
158   PAL0708             6 Gentoo penguin (Pygoscelis papua) Anvers Biscoe
159   PAL0708             7 Gentoo penguin (Pygoscelis papua) Anvers Biscoe
161   PAL0708             9 Gentoo penguin (Pygoscelis papua) Anvers Biscoe
                  Stage Individual.ID Clutch.Completion   Date.Egg
NA                 <NA>          <NA>              <NA>       <NA>
153 Adult, 1 Egg Stage         N31A1               Yes 2007-11-27
155 Adult, 1 Egg Stage         N32A1               Yes 2007-11-27
158 Adult, 1 Egg Stage         N33A2               Yes 2007-11-18
159 Adult, 1 Egg Stage         N34A1               Yes 2007-11-27
161 Adult, 1 Egg Stage         N35A1               Yes 2007-11-27
    Culmen.Length..mm. Culmen.Depth..mm. Flipper.Length..mm. Body.Mass..g.
NA                 NA                NA                  NA            NA
153              46.1              13.2                 211          4500
155              48.7              14.1                 210          4450
158              46.5              13.5                 210          4550
159              45.4              14.6                 211          4800
161              43.3              13.4                 209          4400
       Sex Delta.15.N..o.oo. Delta.13.C..o.oo. Comments
NA    <NA>                NA                NA     <NA>
153 FEMALE           7.99300         -25.51390     <NA>
155 FEMALE           8.14705         -25.46172     <NA>
158 FEMALE           7.99530         -25.32829     <NA>
159 FEMALE           8.24515         -25.46782     <NA>
161 FEMALE           8.13643         -25.32176     <NA>
```

Another important aspect of base R data.frames is that they can utilize `rownames()`. Using `rownames()` is uncommon in workflows from the `tidyverse` or even `data.table`. However, `rownames()` can be very useful when dealing with bioinformatics data. One R package that utilizes `rownames()` extensively is the SummarizedExperiment. `SummarizedExperiments` use rownames and colnames to ensure that data stays coordinated during an analysis.

You can view and set the `rownames()` on a data.frame using the `rownames()` function.

```
# View the current rownames
head(rownames(penguins))
```

```
[1] "1" "2" "3" "4" "5" "6"
```

```
# Set the rownames based on a new value - we'll go over what paste() does shortly
rownames(penguins) <- paste("row_number", 1:nrow(penguins), sep = ".")

# View the new rownames
head(rownames(penguins))
```

```
[1] "row_number.1" "row_number.2" "row_number.3" "row_number.4" "row_number.5"
[6] "row_number.6"
```

Above, we filtered by a column in the data.frame. We can also filter by selecting rownames.

```
penguins[c("row_number.1", "row_number.5"), ]
```

```
             studyName Sample.Number                                Species Region
row_number.1   PAL0708             1 Adelie Penguin (Pygoscelis adeliae) Anvers
row_number.5   PAL0708             5 Adelie Penguin (Pygoscelis adeliae) Anvers
                Island              Stage Individual.ID Clutch.Completion
row_number.1 Torgersen Adult, 1 Egg Stage          N1A1                Yes
row_number.5 Torgersen Adult, 1 Egg Stage          N3A1                Yes
               Date.Egg Culmen.Length..mm. Culmen.Depth..mm.
row_number.1 2007-11-11               39.1              18.7
row_number.5 2007-11-16               36.7              19.3
             Flipper.Length..mm. Body.Mass..g.    Sex Delta.15.N..o.oo.
row_number.1                 181          3750   MALE                NA
row_number.5                 193          3450 FEMALE           8.76651
             Delta.13.C..o.oo.                        Comments
row_number.1                NA Not enough blood for isotopes.
row_number.5         -25.32426                            <NA>
```

Reordering data

We can also change the order of the rows in a data.frame. This is similar to subsetting but instead rearranges the data based on some condition. To reorder the data.frame we can use the `order()` function.

For example, we can order the data by decreasing flipper length

```
head(penguins[order(penguins$Flipper.Length..mm., decreasing = TRUE), ])
```

```
               studyName Sample.Number                           Species Region
row_number.216   PAL0809            64 Gentoo penguin (Pygoscelis papua) Anvers
row_number.154   PAL0708             2 Gentoo penguin (Pygoscelis papua) Anvers
row_number.186   PAL0708            34 Gentoo penguin (Pygoscelis papua) Anvers
row_number.218   PAL0809            66 Gentoo penguin (Pygoscelis papua) Anvers
row_number.228   PAL0809            76 Gentoo penguin (Pygoscelis papua) Anvers
row_number.242   PAL0910            90 Gentoo penguin (Pygoscelis papua) Anvers
               Island              Stage Individual.ID Clutch.Completion
row_number.216 Biscoe Adult, 1 Egg Stage         N19A2               Yes
row_number.154 Biscoe Adult, 1 Egg Stage         N31A2               Yes
row_number.186 Biscoe Adult, 1 Egg Stage         N56A2               Yes
row_number.218 Biscoe Adult, 1 Egg Stage         N20A2               Yes
row_number.228 Biscoe Adult, 1 Egg Stage         N56A2               Yes
row_number.242 Biscoe Adult, 1 Egg Stage         N14A2               Yes
                 Date.Egg Culmen.Length..mm. Culmen.Depth..mm.
row_number.216 2008-11-13               54.3              15.7
row_number.154 2007-11-27               50.0              16.3
row_number.186 2007-12-03               59.6              17.0
row_number.218 2008-11-04               49.8              16.8
row_number.228 2008-11-06               48.6              16.0
row_number.242 2009-11-25               52.1              17.0
               Flipper.Length..mm. Body.Mass..g.  Sex Delta.15.N..o.oo.
row_number.216                 231          5650 MALE           8.49662
row_number.154                 230          5700 MALE           8.14756
row_number.186                 230          6050 MALE           7.76843
row_number.218                 230          5700 MALE           8.47067
row_number.228                 230          5800 MALE           8.59640
row_number.242                 230          5550 MALE           8.27595
               Delta.13.C..o.oo. Comments
row_number.216         -26.84166     <NA>
row_number.154         -25.39369     <NA>
row_number.186         -25.68210     <NA>
```

```
row_number.218         -26.69166     <NA>
row_number.228         -26.71199     <NA>
row_number.242         -26.11657     <NA>
```

R also provides the `sort()` function. See if you can understand the difference between `order()` and `sort()`.

Similar to filtering above, we can also reorder by the `rownames()`.

```
head(penguins[order(rownames(penguins)), ])
```

```
               studyName Sample.Number                                Species
row_number.1     PAL0708             1 Adelie Penguin (Pygoscelis adeliae)
row_number.10    PAL0708            10 Adelie Penguin (Pygoscelis adeliae)
row_number.100   PAL0809           100 Adelie Penguin (Pygoscelis adeliae)
row_number.101   PAL0910           101 Adelie Penguin (Pygoscelis adeliae)
row_number.102   PAL0910           102 Adelie Penguin (Pygoscelis adeliae)
row_number.103   PAL0910           103 Adelie Penguin (Pygoscelis adeliae)
               Region    Island              Stage Individual.ID
row_number.1   Anvers Torgersen Adult, 1 Egg Stage          N1A1
row_number.10  Anvers Torgersen Adult, 1 Egg Stage          N5A2
row_number.100 Anvers     Dream Adult, 1 Egg Stage         N50A2
row_number.101 Anvers    Biscoe Adult, 1 Egg Stage         N47A1
row_number.102 Anvers    Biscoe Adult, 1 Egg Stage         N47A2
row_number.103 Anvers    Biscoe Adult, 1 Egg Stage         N49A1
               Clutch.Completion   Date.Egg Culmen.Length..mm.
row_number.1                 Yes 2007-11-11               39.1
row_number.10                Yes 2007-11-09               42.0
row_number.100               Yes 2008-11-10               43.2
row_number.101               Yes 2009-11-09               35.0
row_number.102               Yes 2009-11-09               41.0
row_number.103               Yes 2009-11-15               37.7
               Culmen.Depth..mm. Flipper.Length..mm. Body.Mass..g.    Sex
row_number.1                18.7                 181          3750   MALE
row_number.10               20.2                 190          4250   <NA>
row_number.100              18.5                 192          4100   MALE
row_number.101              17.9                 192          3725 FEMALE
row_number.102              20.0                 203          4725   MALE
row_number.103              16.0                 183          3075 FEMALE
               Delta.15.N..o.oo. Delta.13.C..o.oo.
row_number.1                  NA                NA
row_number.10            9.13362         -25.09368
```

```
row_number.100           8.97025         -26.03679
row_number.101           8.84451         -26.28055
row_number.102           9.01079         -26.38085
row_number.103           9.21510         -26.22530
                                              Comments
row_number.1          Not enough blood for isotopes.
row_number.10  No blood sample obtained for sexing.
row_number.100                                    <NA>
row_number.101                                    <NA>
row_number.102                                    <NA>
row_number.103                                    <NA>
```

Can you figure out why the data.frame was sorted this way?

Selecting columns of data

Similarly to how we were able to select rows of data from the data.frame, we can also select columns of the data.frame. First, to see what columns are in our data.frame we can use the `colnames()` function.

```
colnames(penguins)
```

```
 [1] "studyName"           "Sample.Number"       "Species"
 [4] "Region"              "Island"              "Stage"
 [7] "Individual.ID"       "Clutch.Completion"   "Date.Egg"
[10] "Culmen.Length..mm."  "Culmen.Depth..mm."   "Flipper.Length..mm."
[13] "Body.Mass..g."       "Sex"                 "Delta.15.N..o.oo."
[16] "Delta.13.C..o.oo."   "Comments"
```

To select columns, we can either specify the columns we want by name

```
head(penguins[, c("studyName", "Species")])
```

```
             studyName                                Species
row_number.1   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.2   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.3   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.4   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.5   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.6   PAL0708 Adelie Penguin (Pygoscelis adeliae)
```

Or by the column index

```
head(penguins[, c(1, 3)])
```

```
             studyName                                 Species
row_number.1   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.2   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.3   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.4   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.5   PAL0708 Adelie Penguin (Pygoscelis adeliae)
row_number.6   PAL0708 Adelie Penguin (Pygoscelis adeliae)
```

If we want to rename a column, we can use the `colnames()` function to reassign the column name with a new name. First, we can get the index of the column we want to change. Then we can reassign that column with the new name.

```
# This returns the index of the colnames() that matches "Species"
which_col_to_change <- which(colnames(penguins) == "Species")

# Rename "Species" to "species"
colnames(penguins)[which_col_to_change] <- "species"
```

One tricky aspect of base R subsetting on data.frames has to do with selecting a single column of data. If you select a single column of data from a data.frame what you get back is a vector. To ensure that you return a data.frame when selecting a single column, add the `drop=FALSE` argument when selecting a single column.

```
# Returns a vector
class(penguins[, "species"])
```

```
[1] "character"
```

```
# Returns a data.frame
class(penguins[, "species", drop = FALSE])
```

```
[1] "data.frame"
```

Modifying columns of data

Adding or removing columns of data to a data.frame is as simple as specifying the name of the column you want to add along with the data that you want to add.

For example, if I wanted to convert the flipper length from mm to m I could multiply each value of flipper length by 0.001.

```
penguins$flipper_length_m <- penguins$Flipper.Length..mm. * 0.001
```

The above data transformation works because of R's *vectorization* rule described above. What if I wanted to define a new variable based on a logical condition? For example, what if I wanted to create a new column based on whether or not a penguin was and Adelie or any other kind? A useful function for performing this action is `ifelse()`. `ifelse()` evaluates a logical condition and returns a value based on whether the condition evaluates to TRUE or FALSE.

```
penguins$is_adelie <- ifelse(
  penguins$species == "Adelie Penguin (Pygoscelis adeliae)",
  "Adelie",
  "Other"
  )
```

If I want to remove a column from the data.frame I can set the value of the column to `NULL`

```
penguins$Flipper.Length..mm. <- NULL
```

String operations

String operations are an important part of data cleaning. Base R supports many functions for transforming strings. Again, these string functions are typically *vectorized* meaning that they can easily be used to modify columns of data.frames.

The "species" column of the penguins data.frame is annoying to work with. We can simplify this column by creating simpler names for each species by excluding the scientific name and the word 'penguin' (we know they're all penguins…)

One base R function we can use to find and replace text is called `gsub()`. `gsub()` takes what is called a regular expression to find text inside of a string and then replaces the text it finds with the text you supply. Regular expressions can become incredibly complex. With this complexity come a lot of power. It would be impossible to teach regular expression in this tutorial. Taking some time to understand the basics of regular expressions can go a long way during data cleaning.

```r
# Replace a space followed by the word 'penguins' and any other character
penguins$species_clean <- gsub(
  pattern = " penguin.*",
  replacement = "",
  x = penguins$species,
  ignore.case = TRUE
  )

# Show the counts for these new categories
table(penguins$species_clean)
```

```
   Adelie Chinstrap    Gentoo
      152        68       124
```

Another base R function that is useful for extracting text is the `substr()` function. `substr()` extracts sub-strings from the supplied text based on the index of the text. For example, to create an even simpler representation of species we could extract only the first letter from the "species" column.

```r
penguins$species_simple <- substr(penguins$species, start = 1, stop = 1)
table(penguins$species_simple)
```

```
  A   C   G
152  68 124
```

A slightly more complicated string operation is extracting text from a string. This involves two functions; one to find a match in the text and another to extract it. Let's say we wanted to extract the word 'penguin' from each of the original species strings. We can do this with a combination of `regmatches()` and `regexpr()`.

```r
penguins$only_penguin <- regmatches(
  x = penguins$species,
  m = regexpr(pattern = "penguin",
              text = penguins$species,
              ignore.case = TRUE)
  )
```

R also supports basic operations on strings like making all characters upper or lowercase. For example, to ensure `only_penguin` contains 'penguin' (and not 'Penguin') we can convert the string to lowercase

```
# lower case
head(tolower(penguins$only_penguin))
```

```
[1] "penguin" "penguin" "penguin" "penguin" "penguin" "penguin"
```

```
# or upper case
head(toupper(penguins$only_penguin))
```

```
[1] "PENGUIN" "PENGUIN" "PENGUIN" "PENGUIN" "PENGUIN" "PENGUIN"
```

Another slightly complicated string operation is string splitting. String splitting in base R can be accomplished using the `strsplit()` function. This function is a little tricky because it returns a list. Here, we split the "species" column on every space character. Using the results from the `strsplit()` function most efficiently involves some advanced R skills.

```
split_species <- strsplit(
  x = penguins$species,
  split = " "
  )

# strsplit returns a list of character vectors split on every space
head(split_species)
```

```
[[1]]
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"

[[2]]
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"

[[3]]
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"

[[4]]
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"

[[5]]
```

```
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"

[[6]]
[1] "Adelie"      "Penguin"     "(Pygoscelis" "adeliae)"
```

```
# To extract the species from this list, we can use a fancy trick
# don't worry about what's happening here right now
penguins$extracted_species <- sapply(split_species, `[[`, 1)
```

Control flow

R contains all of the typical control flow you expect from a programming language.

For-loops are somewhat under-utilized by many R users but can often be the most clear for new (and experienced) users. one common for-loop idion in R is to use a for-loop with the `seq_along()` function. `seq_along()` generates an index of the given vector that you can use to loop over.

```
for (i in seq_along(x)) {
  doSomething(x[i])
}
```

Other looping operations in base R include `while` and `repeat`.

Conditional statements can be tested using if statements. These follow a familiar syntax

```
x <- 10
if (x == 10) {
  print("the value of x is 10")
} else {
  print("the value of x is something else")
}
```

```
[1] "the value of x is 10"
```

If statements in R are *not* vectorized. Use `ifelse()` to perform a vectorized if statement as demonstrated above.

If statements are often combined with for-loops.

```r
# Make some space to save the result
result <- vector("character", length = 10)

# Determine if the number is even or odd
for (i in 1:10) {
  if (i %% 2 == 0) {
    result[i] <- "even"
  } else {
    result[i] <- "odd"
  }
}

result
```

```
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

R also contains a `switch()` statement which can be very useful when programming new functions.

Functions

Eventually you'll want to create your own functions to do stuff. Functions can be defined in R using the following syntax

```r
f <- function(args) {
  expr
}
```

To make this a little more concrete, let's define a function that classifies a penguin based on it's flipper size. Our function will take in a flipper size and a threshold value for determining whether or not the penguin is "large" or "small" based on this flipper size. The function will return the resulting size designation.

```r
determine_size <- function(flipper_size, threshold = 0.2) {
  if (flipper_size >= threshold) {
    size_category <- "large"
  } else {
    size_category <- "small"
  }
```

```
  return(size_category)
}

# Is a 0.5 meter flipper length large or small?
determine_size(flipper_size = 0.5)
```

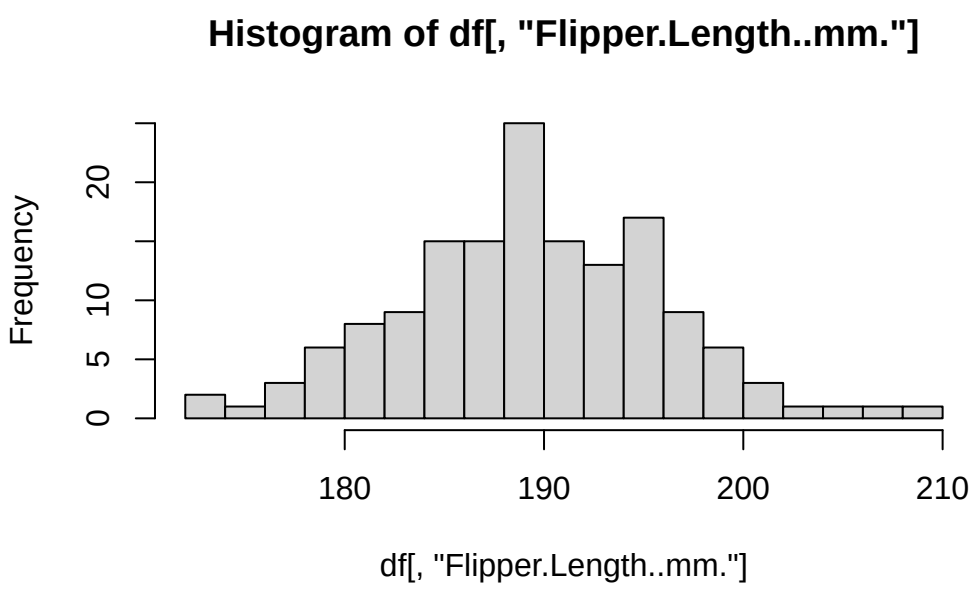
```
[1] "large"
```

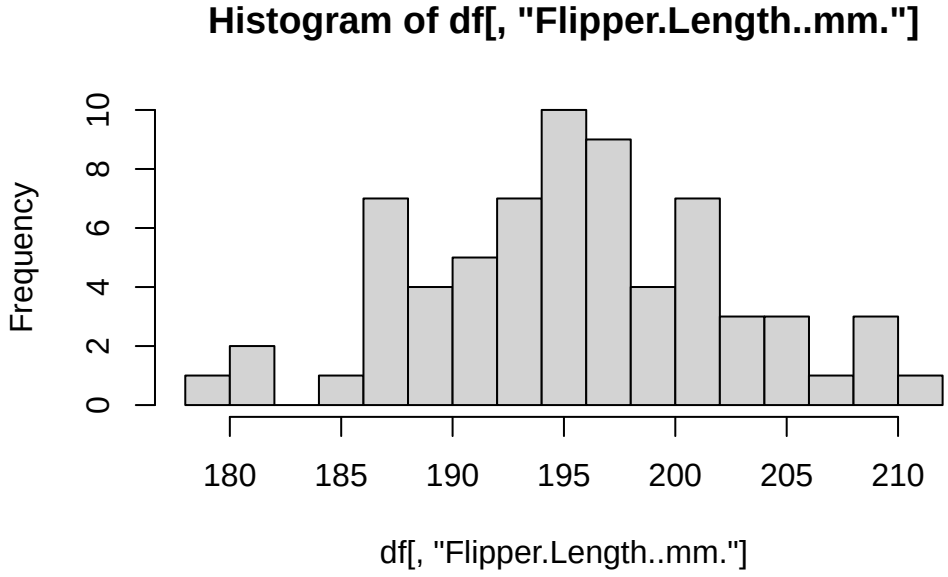
Functional programming

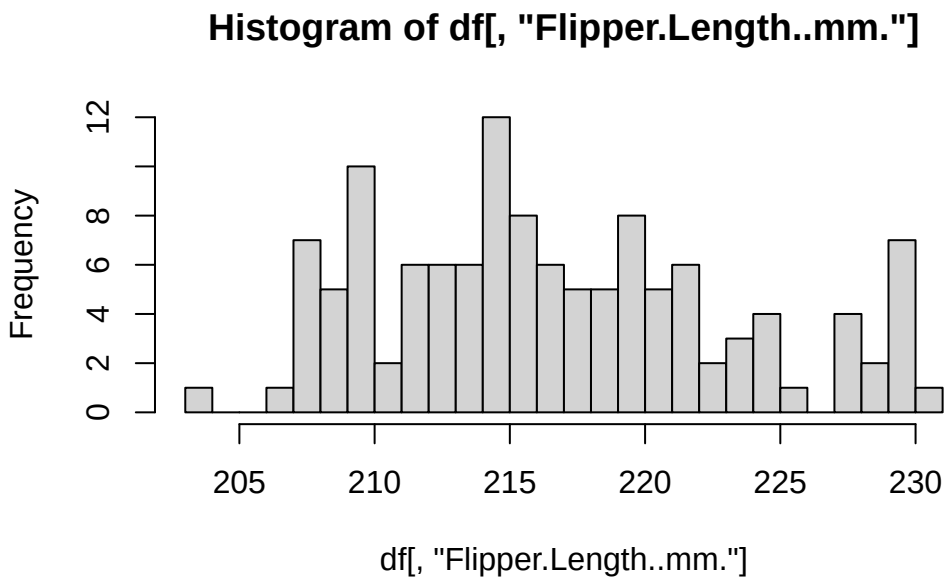
For-loops in R are very simple to understand but most R programmers use functions in the `apply()` family. These functions supply abstractions over common for-loops that make applying functions over lists much easier in R.

The most commonly used functional programming paradigm in R is the use of the `lapply()` function. The `lapply()` function takes a list as input, applies a function to each element of that list, and returns the results as a list. For example, above we split the penguins data.frame into a list of smaller data.frames for each species. Instead of repeating the code to create a histogram from each data.frame, let's define a function that plots a histogram using a data.frame and then apply that function to every data.frame in our "by_species" list of data.frames

```
# Define a function for plotting a histogram of flipper lengths
plot_histogram <- function(df) {
  hist(df[, "Flipper.Length..mm."], breaks=20)
}

# Apply this function to every data.frame
# invisible() is used to hide the output from the hist() function
invisible(
  lapply(X = by_species, FUN = plot_histogram)
  )
```


Histogram of df[, "Flipper.Length..mm."]
Frequency
0
5
10
20
180
190
200
210
df[, "Flipper.Length..mm."]


Histogram of df[, "Flipper.Length..mm."]
Frequency
0
2
4
6
8
10
180
185
190
195
200
205
210
df[, "Flipper.Length..mm."]



R includes some other specialized functions for functional programming. Be sure to explore `apply()` for computing over rows/columns of matrices, `sapply()`and `vapply()` for applying functions over lists and returning vectors, `mapply()` for supplying multiple arguments to a list.

Resources

This only scratches the surface of programming in R. Check out these resources for more information.

- fasteR
- R for Data Science
- Advanced R 2nd Edition
- Advanced R 1st Edition
- The R Inferno

Effective data visualizations

Data visualization is one of the most important skills to develop as a data scientist. We use graphs, instead of tables for instance, to clearly communicate patterns, trends, and comparisons in data in a way that is inherently more interesting and informative than numbers in a grid.

Edward Tufte's, "The Visual Display of Quantitative Information", is probably the most famous text in all of data visualization (I'm guessing). In this text, Tufte outlines a theory of graphics and provides extensive detail regarding techniques for displaying data that maximizes clarity, precision, and efficiency. Tufte describes *graphical excellence* by the following points:

- show the data
- induce the viewer to think about substance…
- avoid distorting what the data have to say
- present many numbers in a small space
- make large data sets coherent
- encourage the eye to compare different pieces of data
- reveal the data at several levels of detail, from a broad overview to the fine structure
- serve a reasonably clear purpose: description, exploration, tabulation, or decoration
- be closely integrated with the statistical and verbal descriptions of the dataset

These points describe *how* to make good visualizations. However, some visualizations are clearly better than others. Take a look at this presentation by Andrew Gelmen and Antony Unwin. Here, they describe what makes some visualizations effective vs. what makes others ineffective and distracting. I think the main conclusion comes down to design vs decoration. Effective data visualizations are intentionally designed to communicate a point in the clearest possible way. Ineffective visualizations often contain clutter in the form of decorations, unnecessary colors, patterns, lines, and unintuitive use of shapes and sizes.

Outside of the design choices that go into creating effective visualizations, there are also other expectations required of graphs. The data points should be labelled so that the viewer immediately knows what they're looking at. The axes should be labelled in a way that's easy to read and do not distort the message. If using colors, the color palette should augment the visualization in a way that enhances the display of information and not only 'looks pretty'.

There are probably thousands (millions?, infinite?) of types of visualizations in use today. However, in scientific communication, there are basically 5 types of graphs that are most commonly used; line graphs, bar graphs, histograms, scatter plots, and pie charts. Most of these have persisted in the literature (with the exception of pie charts) because of their ability to clearly and quickly display visual information about quantitative data. Many other forms of these basic charts exist primarily as variations on a theme but the core display of information remains the same.

Plotting with `ggplot2`

We'll use the ggplot2 R package to start creating effective visualizations. `ggplot2` works in a way that can feel a little strange at first, especially if you're used to creating plots in Python with `matplotlib`, for example. `ggplot2` implements ideas from Leland Wilkinson's, "Grammar of Graphics". `ggplot2`, breaks down a graphic into several key components that can be combined in a layered fashion. These core components typically include:

- **Data**: The dataset you want to visualize.
- **Aesthetics (aes)**: How columns in your data map to visual properties of the graphic. For example, mapping a 'temperature' column to the y-axis, a 'time' column to the x-axis, and a 'city' column to color.
- **Geometries (geoms)**: The visual elements used to represent the data, such as points, lines, bars, histograms, etc.
- **Scales**: Control how the aesthetic mappings translate to visual output (e.g., how data values are converted to colors, sizes, or positions on an axis).
- **Statistics (stats)**: Transformations of the data that are performed before plotting (e.g., calculating a histogram, smoothing a line).
- **Coordinates**: The coordinate system used for the plot (e.g., Cartesian, polar).
- **Faceting**: Creating multiple subplots (a "trellis" or "lattice" plot) based on subsets of the data.
- **Theme**: Non-data elements like background, grid lines, and font choices.

The power of this approach lies in its declarative nature. You specify *what* you want to plot by defining these components, rather than detailing how to draw each element step-by-step (like in `matplotlib`). This makes it easier to build complex visualizations and to switch between different representations of the same data with minimal changes to the code.

General design guidelines for plots

- Don't use a special color for the background color. Use the same color as your presentation background. Generally, either white or black backgrounds are best

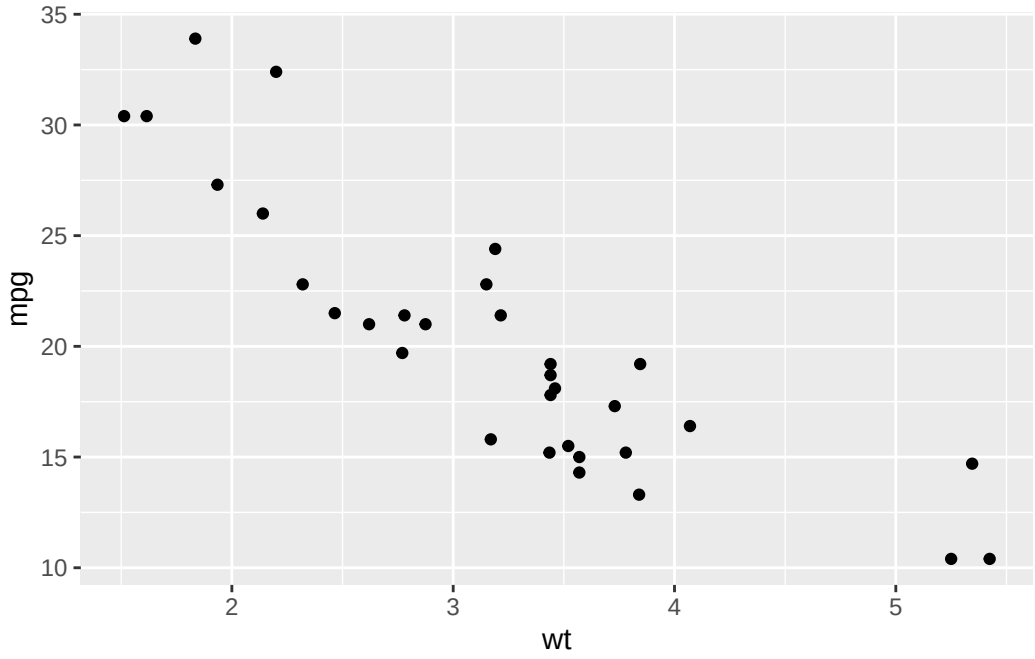
- Axes colors should be high contrast relative to the background. This means black axes on white backgrounds and white axes on black backgrounds.
- Use tick marks that have whole-number intervals. e.g. don't use axis ticks that are 1.33, 2.75, 3.38, ….. Instead, use numbers that people use naturally when counting, e.g. 1, 2, 3 or 2, 4, 6, etc.
- Axes need to be legible. The default font size is often too small. At least 12 point font.
- Axes labels should clearly communicate what is being plotted on each axis. Always show units.
- If showing a plot title, use the title to tell the user what the conclusion is, not simply describing the data on each axis.
- If using multiple colors, they should contrast well with each other. Warm colors (reds, yellows) can be used to emphasize selected data.
- Keep colors consistent across graphs. If you use the color "red" to represent treated samples in one plot, use the color "red" in all other plots where the treated samples are present.
- Use colors to communicate. Hues are not emotionally neutral. A good color palette can affect the audience's perception of the figure (just like a bad one can).
- Only use gridlines if you need them, most of the time they're not needed.
- If using gridlines, be judicious. Their color should be subtle and not obscure the data. Only use grids that matter for the data.
- Choose fonts that are easy to read. sans serif fonts are best. Helvetica or Arial are good default choices
- When drawing error bars, don't make them so wide/large as to obscure the data

A basic ggplot

`ggplot()` constructs a plot from data, what it calls aesthetic mappings, and layers. Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. geoms then determine *how* the data is displayed. The other parts of the ggplot object have been handled automatically (i.e. scales, stats, coordinates, and theme). These, however, can be modified to enhance the plot. Check out the ggplot2 homepage for an overview or the ggplot2 book for details.

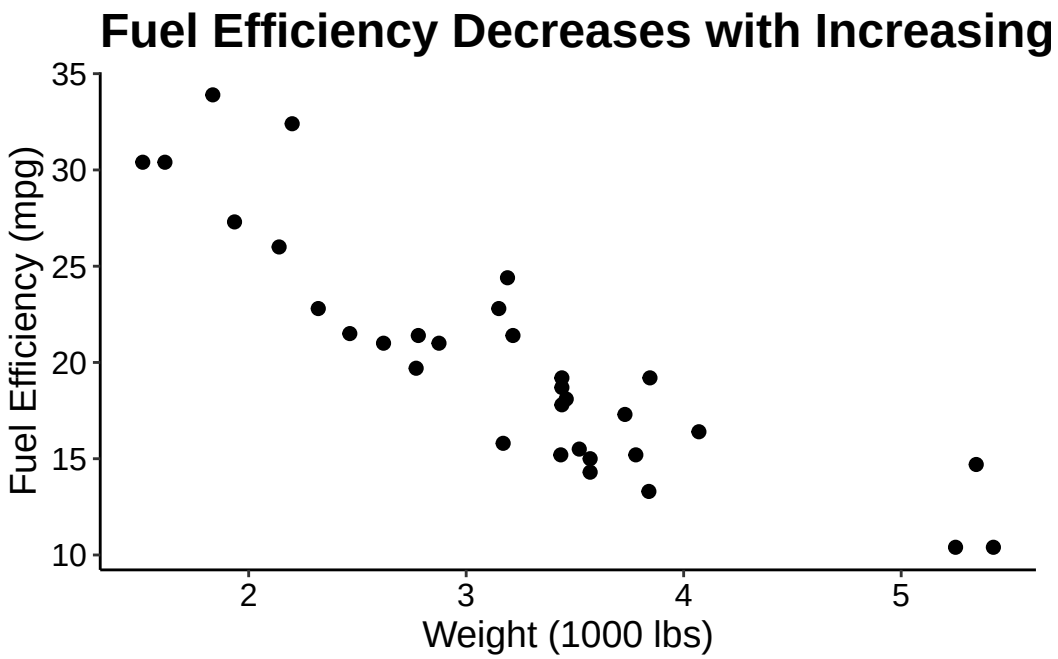
The code below demonstrates the most basic way of creating a plot with `ggplot2`.

```
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg)) +
  geom_point()
```



This plot is okay but can be improved. Let's improve this plot by removing the grey background and gridlines, increasing the font size of the axis ticks, improving the axis labels, and creating an informative title.

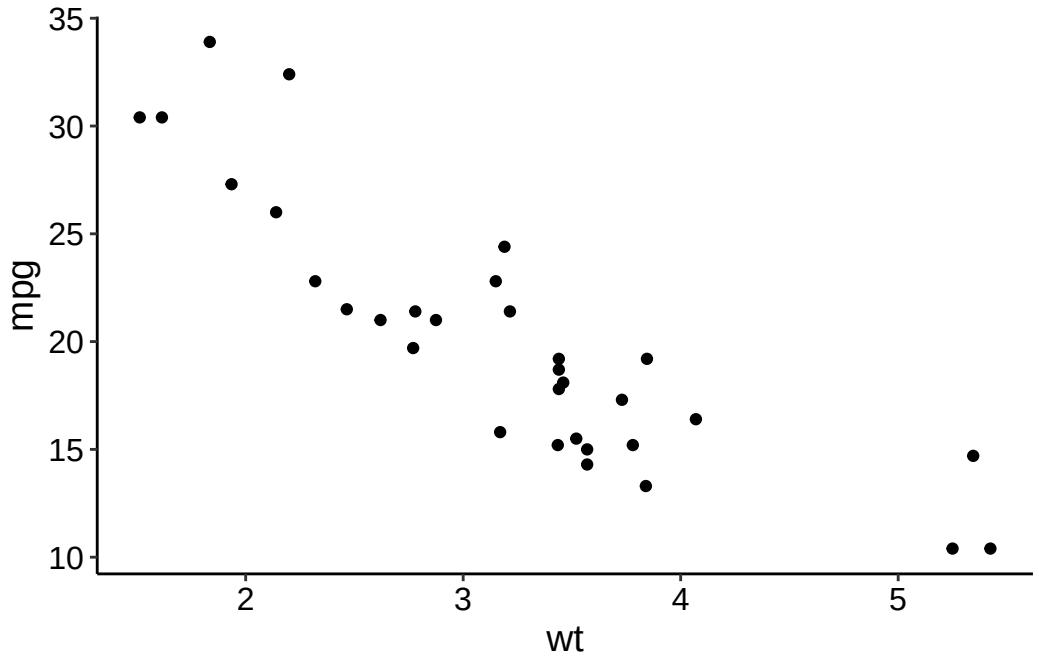
```
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg)) +
  geom_point(size = 2) +        # increase the size of the points
  labs(                         # labs() can be used to modify axis label text
    title = "Fuel Efficiency Decreases with Increasing Weight",
    x  = "Weight (1000 lbs)",
    y = "Fuel Efficiency (mpg)") +
  theme_classic() +             # Removes grey background and gridlines
  theme(                        # Adjust the plot and axes titles, and text
    plot.title = element_text(size = 18, face = "bold", color = "black"),
    axis.title = element_text(size = 14, color = "black"),
    axis.text = element_text(size = 12, color = "black")
  )
```



Saving themes

If you're going to be using the same theme elements often, it can be helpful to save those as a new custom theme. *You don't have to understand exactly how this function works right now - simply modify the arguments to the `theme()` function and figure this out as you improve*

```
theme_clean <- function(...) {
  ggplot2::theme_classic(...) %+replace%
  ggplot2::theme(
      text = ggplot2::element_text(family = "Helvetica"),
      plot.title = ggplot2::element_text(size = 18, face = "bold", color = "black", hjust = 
      axis.title = ggplot2::element_text(size = 14, color = "black"),
      axis.text = ggplot2::element_text(size = 12, color = "black")
    )
}

# The new theme can be applied to other plots
ggplot(data = mtcars, mapping = aes(x = wt, y = mpg)) +
  geom_point() +
  theme_clean()
```

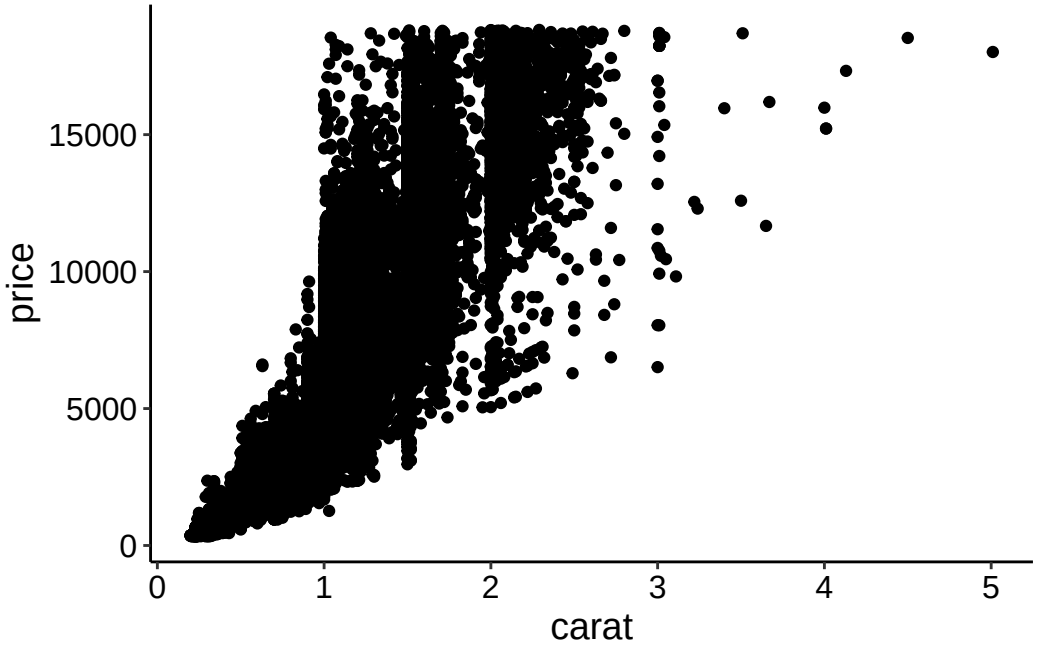


This plot is contains the same information but more quickly and clearly communicates the message by just modifying the design. What other element could be added to this plot to make the trend more apparent?

Scatter plots

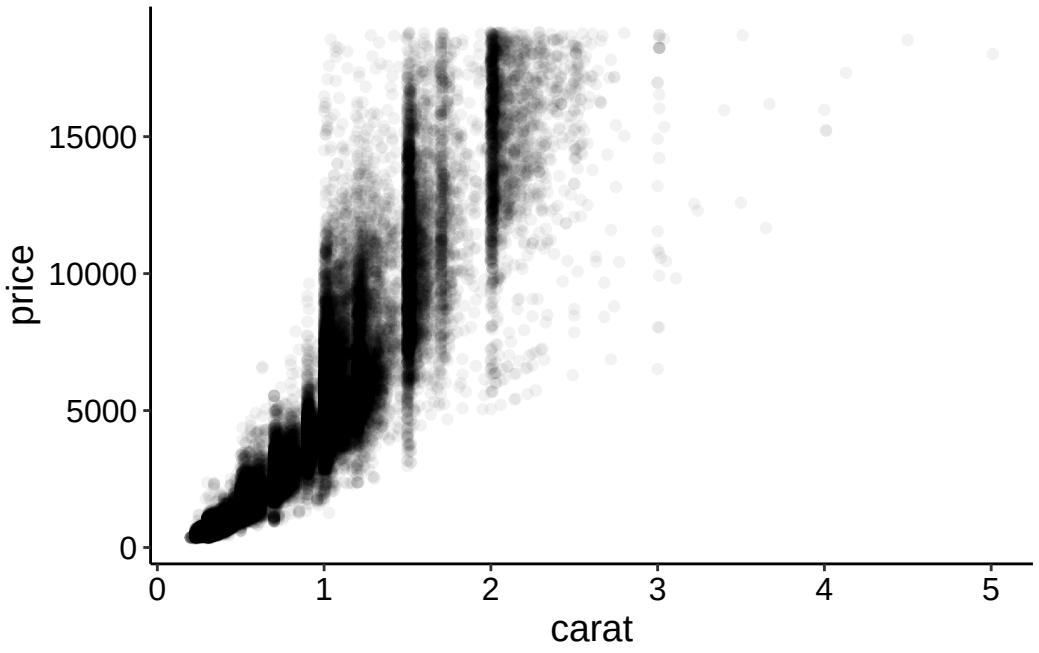
We already showed a basic example of creating a scatter plot above. You can use that as a starting point for generating scatter plots. However, one common issue when designing scatter plots is *overplotting*, or, showing so many points that the data is cluttered. Below is an example of overplotting.

```
ggplot(diamonds, aes(carat, price)) +
  geom_point() +
  theme_clean()
```



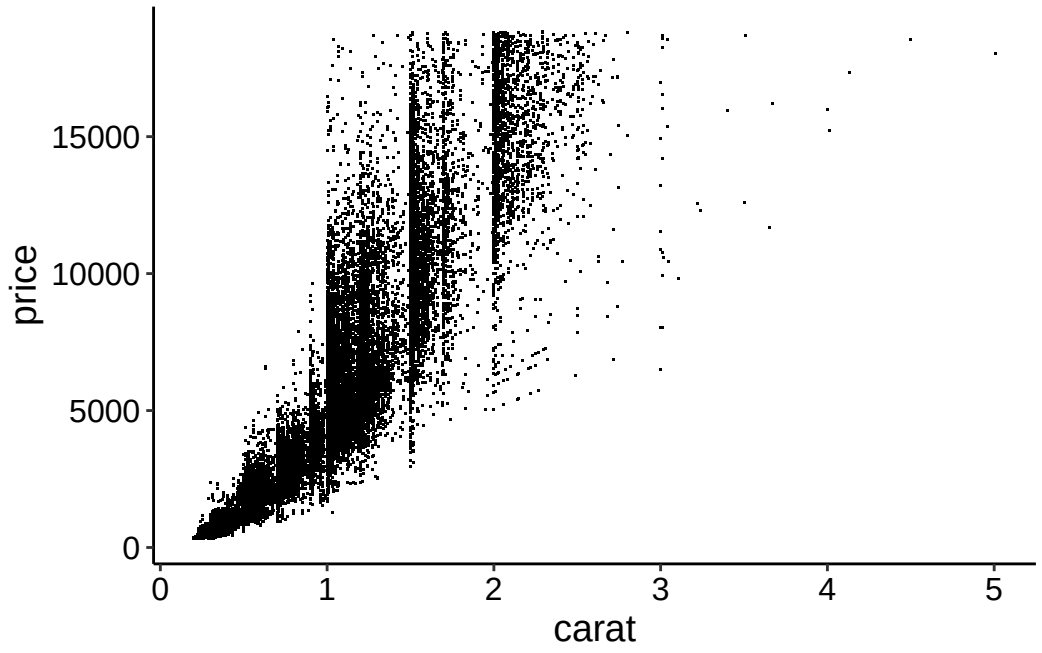
One technique to overcome overplotting is to add transparency to the points

```
ggplot(diamonds, aes(carat, price)) +
  geom_point(alpha = 0.05) +
  theme_clean()
```



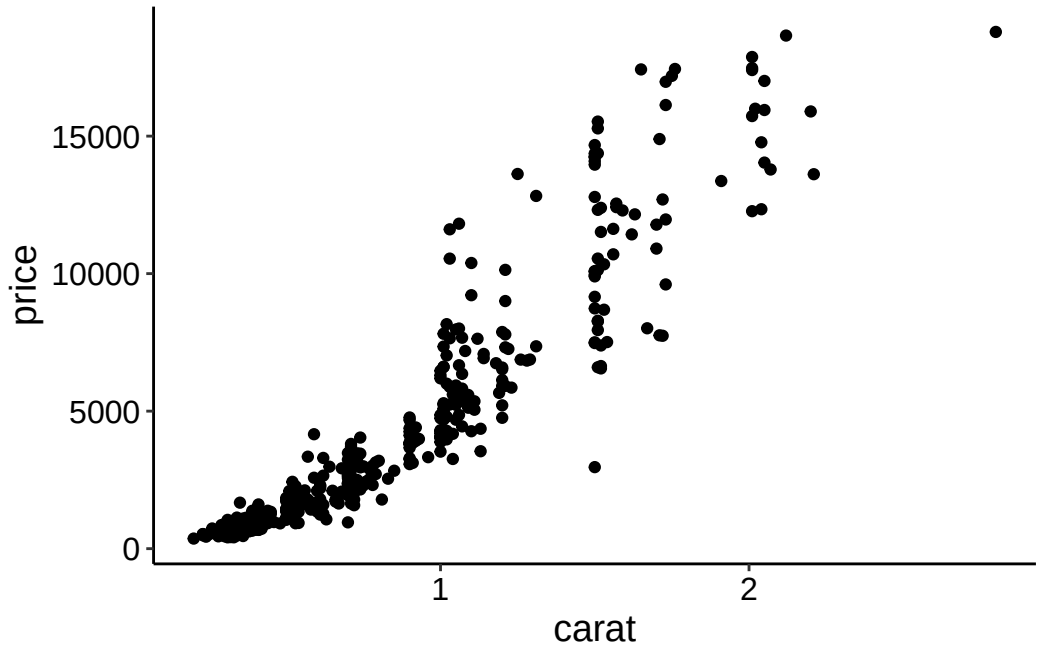
Another is to change the point type. Here, we plot each point as a single dot

```
ggplot(diamonds, aes(carat, price)) +
  geom_point(shape = ".") +
  theme_clean()
```



And another is random subsampling.

```
random_rows <- sample.int(nrow(diamonds), size = 500)

ggplot(diamonds[random_rows, ], aes(carat, price)) +
  geom_point() +
  theme_clean()
```

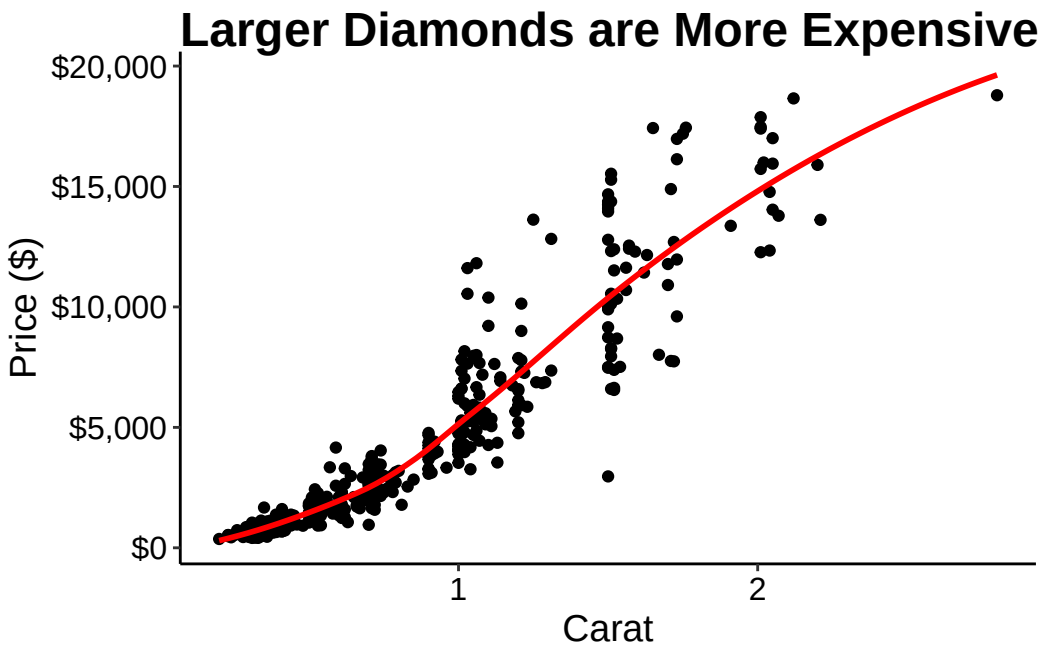


The density of the points could also be summarized. For example, explore `geom_hex()` or `geom_density2d()` geoms.

Let's make a final version of this plot by cleaning up the background, axes, and titles. We'll add a trendline to emphasize the relationship and we'll also use a function to transform the y-axis to dollar format.

```
ggplot(diamonds[random_rows, ], aes(carat, price)) +
  geom_point() +
  geom_smooth(se = FALSE, color = "red") +   # Adds a smooth trendline
  labs(
    title = "Larger Diamonds are More Expensive",
    x  = "Carat",
    y = "Price ($)") +
  scale_y_continuous(             # Formats the y-axis labels as dollar amounts
    labels = scales::dollar_format()
    )  +
  theme_clean()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

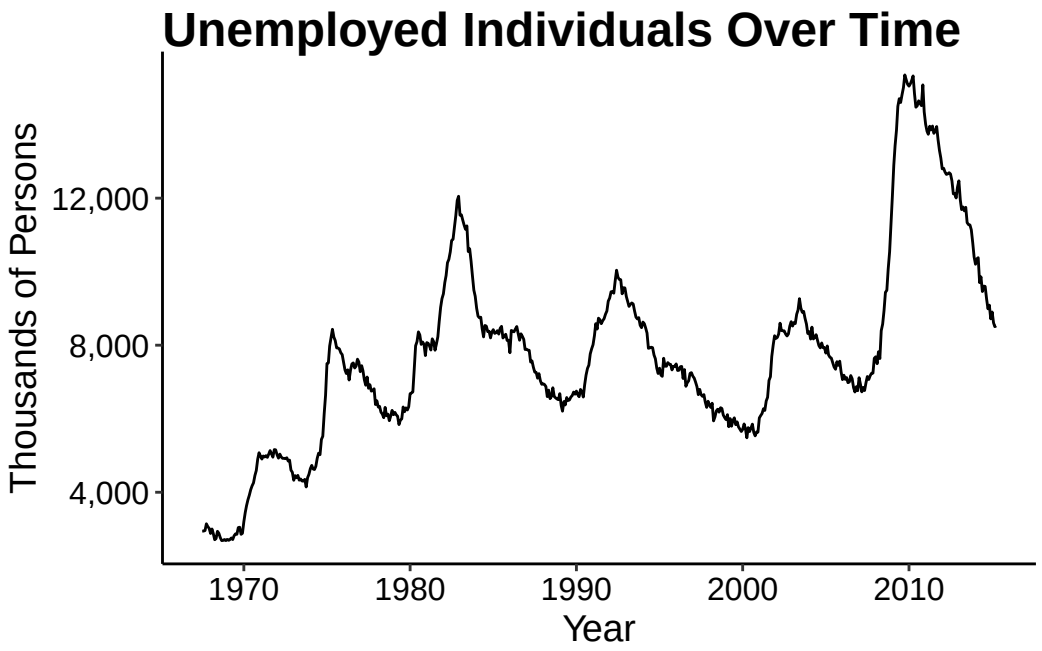


Line graphs

Line graphs are meant to emphasize change in the y-variable over the x-variable. When designing line graphs:

- Take advantage the range of the data. Data should take up about 3/4 of the y-axis. `ggplot` has pretty good defaults for this automatically.
- Choose line weights that do not overshadow points (if present)
- Dashed lines can be hard to read. Use contrasting colors instead
- If presenting two graphs next to each other be sure to match the axes ranges

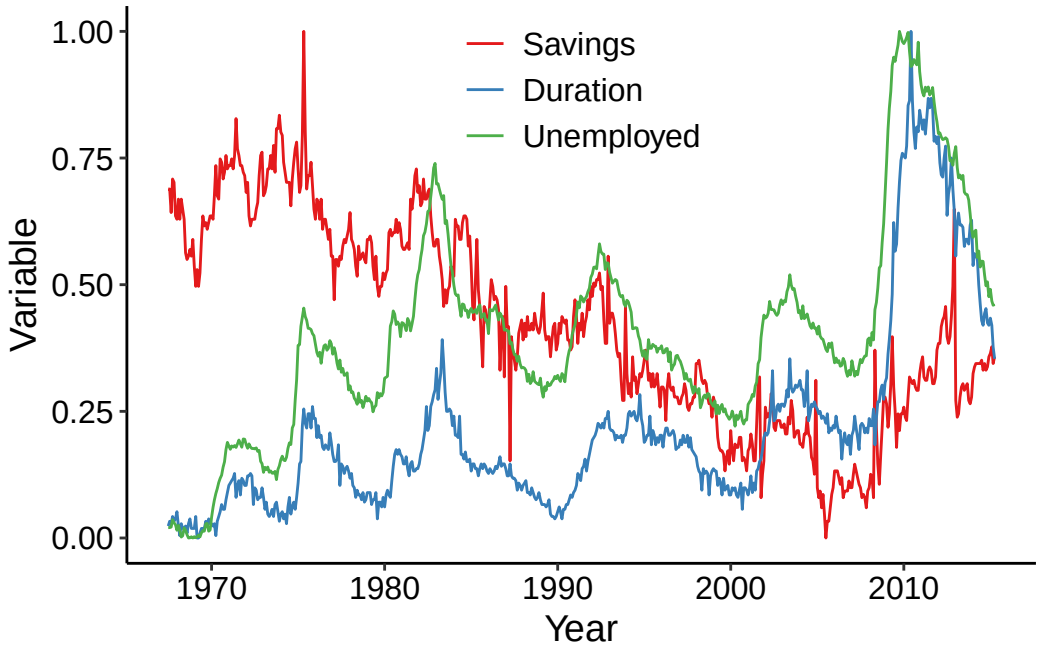
```
ggplot(economics, aes(date, unemploy)) +
  geom_line() +
  labs(
    title = "Unemployed Individuals Over Time",
    x = "Year",
    y = "Thousands of Persons"
    ) +
  scale_y_continuous(labels = scales::number_format(big.mark = ",")) +
  theme_clean()
```



We can also explore adding multiple colors to the plot to compare values. Another useful technique is to add the legend to the plot area.

```
econ_2 <- economics_long[!economics_long$variable %in% c("pce", "pop"), ]

ggplot(econ_2, aes(date, value01, colour = variable)) +
  geom_line() +
  labs(
    x = "Year",
    y = "Variable",
    color = NULL
    ) +
  scale_color_brewer(                              # Add better colors and labels
    palette = "Set1",
    breaks = c("psavert", "uempmed", "unemploy"),
    labels = c("Savings", "Duration", "Unemployed")
    ) +
  scale_y_continuous(labels = scales::number_format(big.mark = ",")) +
  guides(color = guide_legend(position = "inside")) +  # add the legend inside the plot
  theme_clean() +
  theme(
    legend.position.inside = c(0.5, 0.85),      # specify where to put the legend
    legend.text = element_text(size = 12)
```

```
)
```



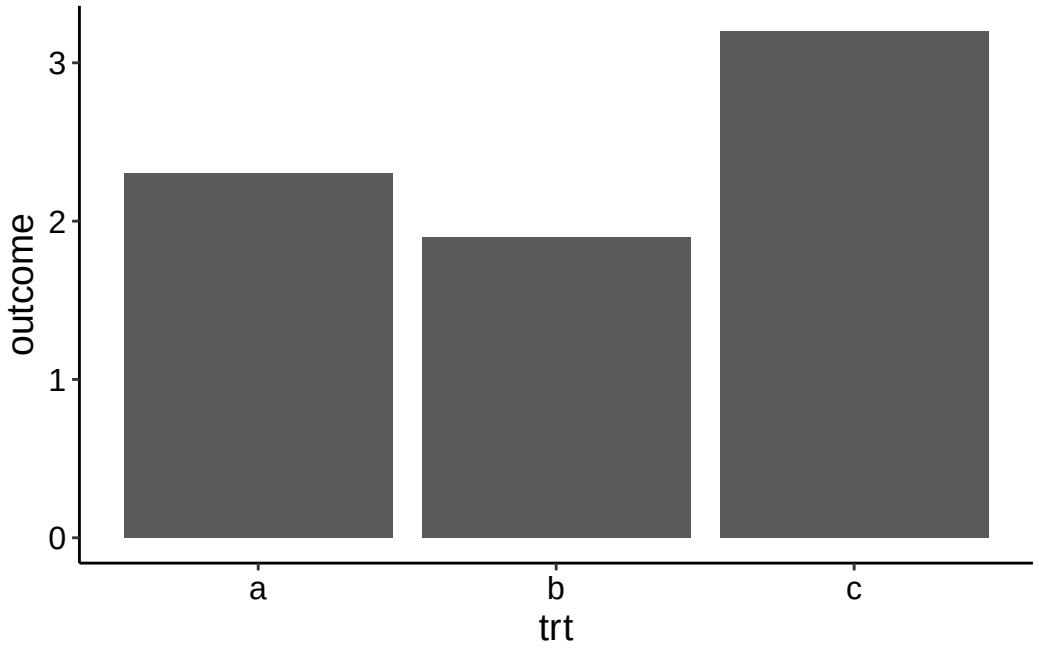
Bar graphs

People generally use bar graphs to display a mean and some variation around the mean. Filled bar plots can also be used to show proportions.

- If there are relatively few data points, consider showing all points and a crossbar for the mean and error instead. For example, as a beeswarm plot. If there are many points, consider a boxplot.
- In most cases, it's usually best to start the y-axis at 0 if the data has a natural 0. The exception is for charts where 0 does not indicate 'nothing' but rather exists on a continuum, for example, temperature in Fahrenheit.
- Don't have bars with too thick of outlines
- Avoid bars that are too thin. Aim for about 1/3 width of the bar as space between bars
- Place extra space between categories if showing bars next to each other
- If showing statistical information try not to overwhelm the data. Use subtle thin lines for comparisons and asterisks for significance.

Activity: Make this example plot better.

```r
df <- data.frame(trt = c("a", "b", "c"), outcome = c(2.3, 1.9, 3.2))

ggplot(df, aes(trt, outcome)) +
  geom_col() +
  theme_clean()
```



Histograms

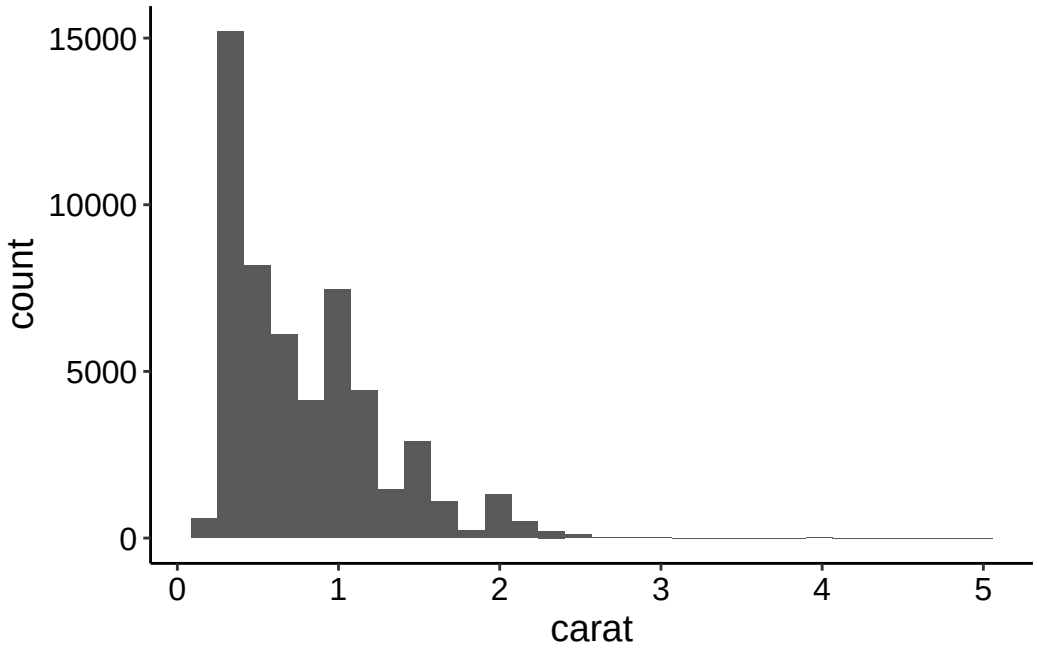
Histograms are used to show distributions of data with their relative frequencies.

- The number of bins influences the interpretation of the data. Play around with the number of bins to ensure the best display
- Don't assign different colors to different bins - it doesn't add any information
- Don't add spaces between bins, as in a bar plot
- If displaying two datasets, you can either overlay each with a different fill and some transparency or split into two histograms with the same y-axis.

Activity: Make this example plot better.

```r
ggplot(diamonds, aes(carat)) +
  geom_histogram() +
  theme_clean()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Saving images

Using `ggplot2` you can save images with the `ggsave()` function. `ggsave()` can automatically detect the image format by the file extension. The `ggsave()` function works by saving the last plot created.

```
ggplot(data, aes(x, y)) +
  geom_point()

ggsave("my-pretty-plot.pdf", width = 8, height = 6)
```

If using base R, you typically open the graphics device (`png()`, `pdf()`, `jpeg()`, etc.) first, depending on the file format you want to save, then create the plot, and close the plot device.

```
pdf("another-pretty-plot.pdf", width = 8, height = 6)
plot(x, y)
dev.off()
```

Image file formats

- Save your images in .pdf format or .svg format. These formats are called vector graphics. Vector graphics are infinitely scalable. They never lose resolution no matter the zoom level.
- Vector graphics don't play nicely with Word documents. If you need to share images that will be used in Word docs, save your image as a high-quality .png. .png files are raster graphics. Raster based images store actual pixel values so they cannot be infinitely zoomed. They lose quality at high magnification. Consider at least a dpi of 300 when saving .pngs.
- Vector graphics are also editable using a program like Inkscape. Often, you may need to add some custom color or annotations using an illustrator program. Vector graphics allow you to do this.

Resources

- Learning ggplot
- ggplot2 Book
- Modern Data Visualization with R
- R for Data Science Data Viz chapter
- Fundamentals of Data Visualization
- The Visual Display of Quantitative Information
- R Graph Gallery

Scientific presentations

A science presentation is more than just displaying your results on a slide deck. Good content doesn't speak for itself. A good science presentation is more like a **story** that weaves together a problem (gap in knowledge), a logical thought process for solving that problem (hypothesis), data that supports your claims (figures), and a compelling delivery. The structure of your presentation, the design of the slides, the flow of information from one slide to the next, and your delivery of that information, are all aspects that your need to consider if you want to create informative and interesting presentations. Like everything else in life, this takes time and practice. Below are a few tips that I have learned. For a good book on this topic check out Designing Science Presentations by Matt Carter.

General tips

- Keep it simple. Well designed presentations translate complex information into simple messages. Richard Feyman was famous for this ability.
- Subtract! It's easy to keep adding information, text, images, etc. to a presentation. It's much harder to know what information to take away. Generally, try to deliberately take away any elements of your presentation that don't add value. Simple is not boring, it's focused.
- Your presentation is about the audience. Determine who your audience is and what they are likely to know or not know. Design all aspects of your presentation with this in mind.
- Learn some principles of design and apply them to your presentations. Design is not decoration. Collect examples of good presentations and take notes from people who do it well.
- Color is a powerful tool for communication. Intentional color choices can draw attention to important data, emphasize points, and evoke emotions. Poorly chosen colors can be distracting and make your content unreadable
- Font choices matter. Fonts can convey tone and personality. Use a font that increases readability. Use italics or bolding to emphasize important ideas.
- Words matter. Choose words that express your ideas precisely, concisely, and as clearly as possible.
- There's no correct number of slides. It comes down to your presentation and delivery.
- Everything doesn't need to be shown - sometimes the best way of communicating an idea is by narration alone.

Slide presentation tips

Generally, you'll want to design a slide presentation in a way that focuses on overall ideas rather than just moving from one slide to the next. Think about the structure of your presentation. This generally means following the scientific method.

For your internship presentations, the structure can be something like:

1. Introduction / background
    - *What is the context and why should we care about it?.*
    - What does the audience need to know in order to understand your conclusions later?
    - Importantly, lead the audience in understanding the gap in knowledge. **What problem are you trying to solve?**. Clearly emphasize your goal.
    - Think about the introduction like a funnel. Start with a broad topic and then narrow in on your what your focus will be for the remainder of the presentation. General –> Specific.
2. Experimental methods
    - Clearly explain *how* you're going to solve the problem. What data you used, and what methods.
    - Explain *why* you chose the methods you did and *how* they answer parts of your overall hypothesis.
    - Don't forget to mention the things that your methods lack.
3. Results
    - Take the audience through your data step-by-step, starting with the simplest results and building into more complex ideas.
    - Don't show too much on one slide. One (or two) figures is usually all you need.
    - Don't put anything on a slide that you aren't going to discuss.
    - **Explain your figures** if you put it on the slide, explain it. Tell the audience what the graph is showing, explain to them the conclusions you draw from it.
    - Be sure to connect your data to the larger story of your presentation - how does this data relate to the information you talked about earlier?
4. Discussion / conclusions
    - Specific –> General
    - Recap the original problem.
    - Recap your methods and why they are appropriate to solve this problem.
    - Pull out the main points and emphasize them. Tell the audience why they are important.
    - Mention any gaps or future work.

- Conclude on broader implications.

In addition to this structure, sometimes it's good to include some measure of progress in the form of slide numbers or a "home" slide - a slide that you come back to that is used to orient the audience to their place in the presentation.

Example

Here is a link to my PhD proposal slides presentation. I tried to incorporate good design principles when creating this. While it's certainly not perfect and I'm certainly not the best presenter, here are a few of the design choices I tried to make to improve the presentation.

- slide 1: eye-catching background with presenter information in contrasting white font. Note that the same font is used throughout the presentation.
- slide 2: High contrast slides, modern black font (Helvetica Neue) on white inconspicuous background. Relatively few words on the slide. Give the audience an idea of the structure of the presentation.
- slide 3: Section header slide used to orient the audience. Section headers can be a good way to give structure to a presentation.
- slides 4-8: General information to specific problem. What are TEs? Why are they important for cancer? How do they contribute to immunotherapy response? Why is this difficult to study? General –> Specific.
- slide 9: An entire slide dedicated to emphasizing the main point of the rest of the proposal.
- slide 11: The use of bolding in the bullet points to emphasize the main point of each. Layout of the slide is designed as question on the left and answer on the right in italics.
- slide 17-16: Results slides showing limited number of figures on each slide with clear data and axes labels. Each title reflects what I want the audience to take away from the figures.
- slide 22: Use color to emphasize the main paper of interest.
- slide 23: Repeat the question and answer structure
- slide 39: Conclusions recap the entire proposal. Bold text is used to emphasize where this proposal falls into the overall scientific goals.

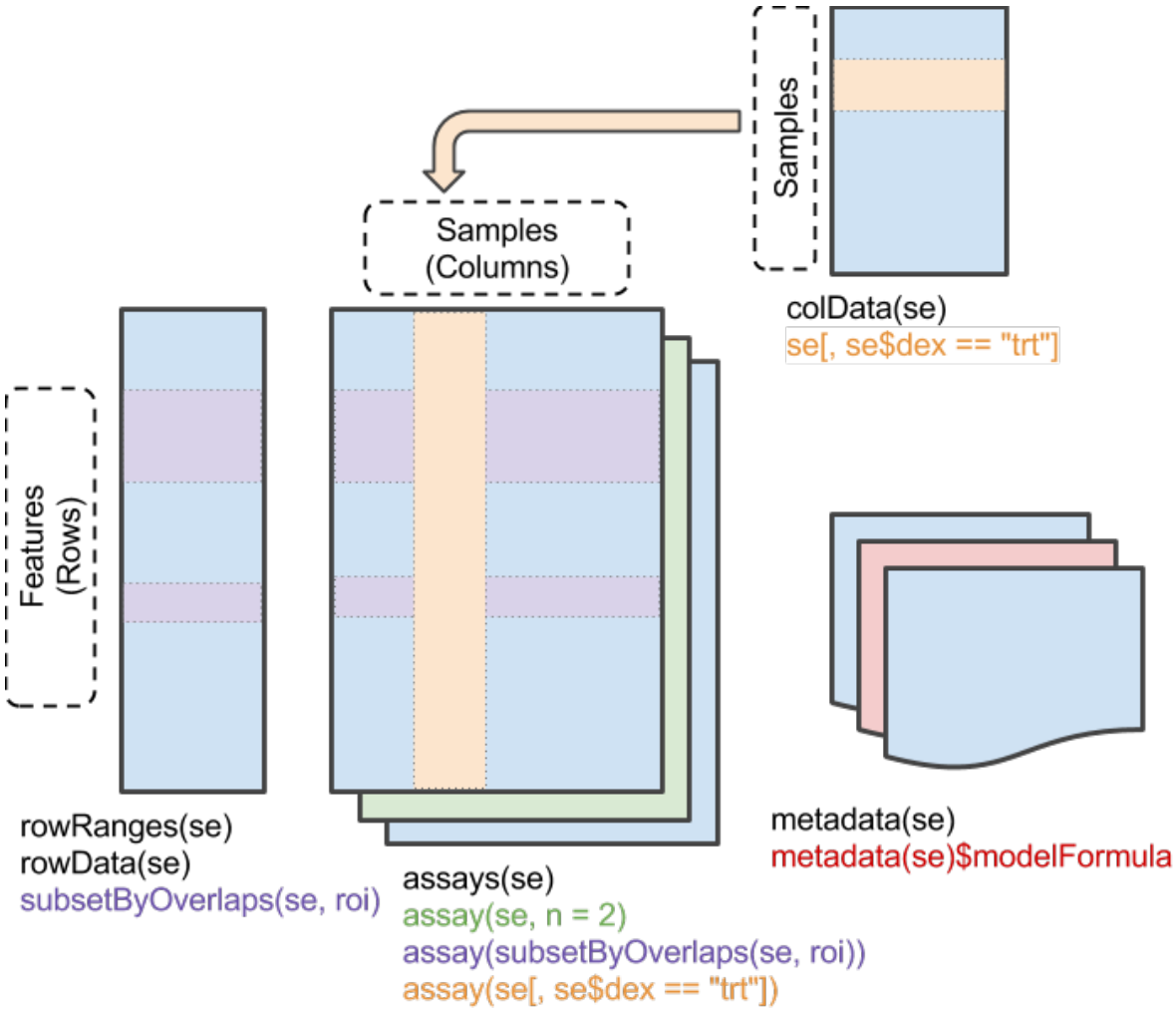
Practical introduction to SummarizedExperiments

What are `SummarizedExperiments`

`SummarizedExperiments` are R objects meant for organizing and manipulating rectangular matrices that are typically produced by arrays or high-throughput sequencing. If you are doing any kind of analysis that requires associating feature-level data (RNA-seq gene counts, methylation array loci, ATAC-seq regions, etc.) with the genomic coordinates of those features and the sample-level metadata with which those features were measured, then you should be using a `SummarizedExperiment` to organize, manipulate, and store your results.

Please take a moment to read through the first 2 sections (at least) of the SummarizedExperiment vignette in order to familiarize yourself with what `SummarizedExperiments` are and their structure. I will demonstrate *how* you can use `SummarizedExperiments` below.

From the `SummarizedExperiment` vignette:



The `SummarizedExperiment` object coordinates four main parts:

- `assay()`, `assays()`: A matrix-like or list of matrix-like objects of identical dimension
  - matrix-like: implements `dim()`, `dimnames()`, and 2-dimensional `[`, `[<-` methods.
  - rows: genes, genomic coordinates, etc.
  - columns: samples, cells, etc.
- `colData()`: Annotations on each column, as a DataFrame.
  - E.g., description of each sample
- `rowData()` and / or `rowRanges()`: Annotations on each row.
  - `rowRanges()`: coordinates of gene / exons in transcripts / etc.
  - `rowData()`: P-values and log-fold change of each gene after differential expression analysis.

- `metadata()`: List of unstructured metadata describing the overall content of the object.

In order to better understand how they work, let's construct a `SummarizedExperiment` from scratch.

Constructing a `SummarizedExperiment`

Hopefully you'll already be working with data that is in a `SummarizedExperiment` or some other class that derives from one. But just in case you don't have data structured as a `SummarizedExperiment` it's useful and instructive to understand how to create one from scratch.

To be most useful, a `SummarizedExperiment` should have at least:

- A matrix of data with features in rows and samples in columns
- A metadata data.frame with samples as rownames and columns describing their properties

Another really useful object to add to `SummarizedExperiments` is a GRanges object describing the genomic locations of each feature in the matrix. Adding this to the `SummarizedExperiment` creates what is called a `RangedSummarizedExperiment` that acts just like a regular `SummarizedExperiment` with some extra features.

To construct our basic `SummarizedExperiment`:

- We'll create a 'counts' matrix with gene IDs as rows and Samples in columns
- We'll add some metadata describing the Samples
- We'll add on `GRanges()` describing the genomic location of the genes

Construct the counts matrix

```
suppressPackageStartupMessages(library(SummarizedExperiment))


counts <- matrix(
  data = rnbinom(n = 200 * 6, mu = 100, size = 1 / 0.5),
  nrow = 200,
  dimnames = list(paste0("Gene", 1:200), paste0("Sample", 1:6))
)

# Take a peek at what this looks like
counts[1:5, 1:5]
```

```
      Sample1 Sample2 Sample3 Sample4 Sample5
Gene1      51      22     206     167      76
Gene2      78      78      36      42     116
Gene3     181     262      35     108     243
Gene4     189      24      42     190      30
Gene5      77     192     140      91     211
```

Construct the sample metadata

It is important that the sample metadata be either a `data.frame` or a `DataFrame` object because `SummarizedExperiment` requires the `colData()` to have rownames that match the colnames of the count matrix.

```
coldata <- data.frame(
  SampleName = colnames(counts),
  Treatment = gl(2, 3, labels = c("Control", "Treatment")),
  Age = sample.int(100, 6),
  row.names = colnames(counts)
)

# Take a peek at what this looks like
coldata
```

```
        SampleName Treatment Age
Sample1    Sample1   Control  95
Sample2    Sample2   Control  61
Sample3    Sample3   Control  54
Sample4    Sample4 Treatment  69
Sample5    Sample5 Treatment  12
Sample6    Sample6 Treatment  40
```

Notice that all of the rownames of the metadata are in the same order as the colnames of the counts matrix. **This is necessary**.

Construct gene range annotations

You will usually have gene annotations or `GRanges` objects loaded from a GTF file or you may even create `GRanges` yourself by specifying the chromosome, start, end, and strand, information manually.

```r
rowranges <- GRanges(
  rep(c("chr1", "chr2"), c(50, 150)),
  IRanges(floor(runif(200, 1e5, 1e6)), width = 100),
  strand = sample(c("+", "-"), 200, TRUE),
  feature_id = sprintf("ID%03d", 1:200),
  gene_type = sample(c("protein_coding", "lncRNA", "repeat_element"), 200, replace = TRUE)
)
names(rowranges) <- rownames(counts)

# Take a peek at what this looks like
rowranges
```

```
GRanges object with 200 ranges and 2 metadata columns:
          seqnames        ranges strand |  feature_id      gene_type
             <Rle>     <IRanges>  <Rle> | <character>    <character>
    Gene1     chr1 912824-912923      - |       ID001 repeat_element
    Gene2     chr1 854941-855040      - |       ID002 repeat_element
    Gene3     chr1 478312-478411      + |       ID003         lncRNA
    Gene4     chr1 902749-902848      - |       ID004 protein_coding
    Gene5     chr1 526446-526545      + |       ID005 protein_coding
      ...      ...            ...    ... .         ...            ...
  Gene196     chr2 828843-828942      + |       ID196         lncRNA
  Gene197     chr2 725643-725742      - |       ID197 protein_coding
  Gene198     chr2 580749-580848      + |       ID198 protein_coding
  Gene199     chr2 404310-404409      + |       ID199         lncRNA
  Gene200     chr2 303121-303220      - |       ID200         lncRNA
  -------
  seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Construct the `SummarizedExperiment` object

With these pieces of information we're ready to create a `SummarizedExperiment` object.

```r
se <- SummarizedExperiment(
  assays = list(counts = counts),
  colData = coldata,
  rowRanges = rowranges
)

# Printing the object gives a summary of what's inside
se
```

```
class: RangedSummarizedExperiment
dim: 200 6
metadata(0):
assays(1): counts
rownames(200): Gene1 Gene2 ... Gene199 Gene200
rowData names(2): feature_id gene_type
colnames(6): Sample1 Sample2 ... Sample5 Sample6
colData names(3): SampleName Treatment Age
```

Accessing parts of the `SummarizedExperiment` object

Every part of the `SummarizedExperiment` object can be extracted with its accessor function. To extract a particular assay you can use the `assay()` function. To extract the column metadata you can use the `colData()` function. To extract the GRanges for the rows of the matrix you can use the `rowRanges()` function. The `rowData()` function also allows you to access row-level annotation information from data added to the `rowData` slot or by the `mcols()` of the `rowRanges`. This will be made more clear below.

Getting the count matrix

```
assay(se, "counts") |> head()
```

```
      Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
Gene1      51      22     206     167      76     161
Gene2      78      78      36      42     116     135
Gene3     181     262      35     108     243     298
Gene4     189      24      42     190      30     212
Gene5      77     192     140      91     211      86
Gene6      51      57      21     141      14     190
```

To see what assays are available you can use the `assays()` function

```
assays(se)
```

```
List of length 1
names(1): counts
```

Getting the column metadata

```
colData(se)
```

```
DataFrame with 6 rows and 3 columns
         SampleName Treatment       Age
        <character>  <factor> <integer>
Sample1     Sample1 Control          95
Sample2     Sample2 Control          61
Sample3     Sample3 Control          54
Sample4     Sample4 Treatment        69
Sample5     Sample5 Treatment        12
Sample6     Sample6 Treatment        40
```

Getting the rowRanges

```
rowRanges(se)
```

```
GRanges object with 200 ranges and 2 metadata columns:
          seqnames        ranges strand |  feature_id      gene_type
             <Rle>     <IRanges>  <Rle> | <character>    <character>
    Gene1     chr1 912824-912923      - |       ID001 repeat_element
    Gene2     chr1 854941-855040      - |       ID002 repeat_element
    Gene3     chr1 478312-478411      + |       ID003         lncRNA
    Gene4     chr1 902749-902848      - |       ID004 protein_coding
    Gene5     chr1 526446-526545      + |       ID005 protein_coding
      ...      ...           ...    ... .         ...             ...
  Gene196     chr2 828843-828942      + |       ID196         lncRNA
  Gene197     chr2 725643-725742      - |       ID197 protein_coding
  Gene198     chr2 580749-580848      + |       ID198 protein_coding
  Gene199     chr2 404310-404409      + |       ID199         lncRNA
  Gene200     chr2 303121-303220      - |       ID200         lncRNA
  -------
  seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Getting the rowData

Note that rowData in this case is the same as `mcols()` of the `rowRanges`

```
rowData(se)
```

```
DataFrame with 200 rows and 2 columns
         feature_id      gene_type
        <character>    <character>
Gene1         ID001 repeat_element
Gene2         ID002 repeat_element
Gene3         ID003         lncRNA
Gene4         ID004 protein_coding
Gene5         ID005 protein_coding
...             ...            ...
Gene196       ID196         lncRNA
Gene197       ID197 protein_coding
Gene198       ID198 protein_coding
Gene199       ID199         lncRNA
Gene200       ID200         lncRNA
```

Modifying a SummarizedExperiment

Once you create a SummarizedExperiment you are not stuck with the information in that object. SummarizedExperiments allow you to add and modify the data within the object.

Adding assays

For example, we may wish to calculate counts per million values for our counts matrix and add a new assay back into our SummarizedExperiment object.

```
# Calculate counts per million
counts <- assay(se, "counts")
cpm <- counts / colSums(counts) * 1e6

# Add the CPM data as a new assay to our existing se object
assay(se, "cpm") <- cpm

# And if we wish to log-scale these values
assay(se, "logcounts") <- log2(cpm)

# Now there are three assays available
assays(se)
```

```
List of length 3
names(3): counts cpm logcounts
```

Note: Instead of creating intermediate variables we could also directly use the assays like so:

```
assay(se, "cpm") <- assay(se, "counts") / colSums(assay(se, "counts")) * 1e6
```

Adding metadata

`SummarizedExperiment` objects use the `$` to get and set columns of the metadata contained in the `colData` slot. For example, to get all of the Ages we can use:

```
se$Age
```

```
[1] 95 61 54 69 12 40
```

If we want to add a new column we simply create the new column in the same way

```
se$Batch <- factor(rep(c("A", "B", "C"), 2))

# Now you can se that a new 'Batch` column has been added to the colData
colData(se)
```

```
DataFrame with 6 rows and 4 columns
         SampleName Treatment       Age    Batch
        <character>  <factor> <integer> <factor>
Sample1     Sample1 Control          95        A
Sample2     Sample2 Control          61        B
Sample3     Sample3 Control          54        C
Sample4     Sample4 Treatment        69        A
Sample5     Sample5 Treatment        12        B
Sample6     Sample6 Treatment        40        C
```

Adding rowData

We can also modify the data which describes each feature in the matrix by adding columns to the rowData. For example, let's create a new column called `Keep` if the gene is a protein_coding gene.

```
rowData(se)$Keep <- rowData(se)$gene_type == "protein_coding"

rowData(se)
```

```
DataFrame with 200 rows and 3 columns
         feature_id      gene_type      Keep
        <character>    <character> <logical>
Gene1         ID001 repeat_element     FALSE
Gene2         ID002 repeat_element     FALSE
Gene3         ID003         lncRNA     FALSE
Gene4         ID004 protein_coding      TRUE
Gene5         ID005 protein_coding      TRUE
...             ...            ...       ...
Gene196       ID196         lncRNA     FALSE
Gene197       ID197 protein_coding      TRUE
Gene198       ID198 protein_coding      TRUE
Gene199       ID199         lncRNA     FALSE
Gene200       ID200         lncRNA     FALSE
```

Subsetting `SummarizedExperiment` objects

`SummarizedExperiments` follow the basic idea of

```
se[the rows you want, the columns you want]
```

With a `SummarizedExperiment` "the rows you want" corresponds to the features in the rows of the matrix/rowData and "the columns you want" corresponds to the metadata in `colData`

Subsetting based on sample metadata

For example, if we want to select all of the data belonging only to samples in the Treatment group we can use the following:

```
(trt <- se[, se$Treatment == "Treatment"])
```

```
class: RangedSummarizedExperiment
dim: 200 3
metadata(0):
```

```
assays(3): counts cpm logcounts
rownames(200): Gene1 Gene2 ... Gene199 Gene200
rowData names(3): feature_id gene_type Keep
colnames(3): Sample4 Sample5 Sample6
colData names(4): SampleName Treatment Age Batch
```

Notice how the `dim` of the object changed from 6 to 3. This is because we have selected only the Samples from the original `SummarizedExperiment` object from the treatment group. The cool thing about `SummarizedExperiments` is that all of the assays have also been subsetted to reflect this selection!

Take a look at the "logcounts" assay. It only contains Samples 4, 5, and 6.

```
assay(trt, "logcounts") |> head()
```

```
       Sample4   Sample5  Sample6
Gene1 12.98955 11.820364 12.98222
Gene2 11.01370 12.519327 12.62367
Gene3 12.32732 13.576118 13.82501
Gene4 13.23120 10.453743 13.34930
Gene5 12.15910 13.326941 11.99870
Gene6 12.68640  9.428737 13.23120
```

Of course you can combine multiple conditions as well

```
se[, se$Batch %in% c("B", "C") & se$Age > 10]
```

```
class: RangedSummarizedExperiment
dim: 200 4
metadata(0):
assays(3): counts cpm logcounts
rownames(200): Gene1 Gene2 ... Gene199 Gene200
rowData names(3): feature_id gene_type Keep
colnames(4): Sample2 Sample3 Sample5 Sample6
colData names(4): SampleName Treatment Age Batch
```

Subsetting based on rows

We can also select certain features that we want to keep using row subsetting. For example to select only the first 50 rows:

```
se[1:50, ]
```

```
class: RangedSummarizedExperiment
dim: 50 6
metadata(0):
assays(3): counts cpm logcounts
rownames(50): Gene1 Gene2 ... Gene49 Gene50
rowData names(3): feature_id gene_type Keep
colnames(6): Sample1 Sample2 ... Sample5 Sample6
colData names(4): SampleName Treatment Age Batch
```

Notice how the `dim` changed from 200 to 50 reflecting the fact that we have only selected the first 50 rows.

This subsetting is very useful when combined with logical operators. Above we created a vector in `rowData` called "Keep" that contained TRUE if the corresponding row of the se object was a coding gene and FALSE otherwise. Let's use this vector to subset our se object.

```
(coding <- se[rowData(se)$Keep, ])
```

```
class: RangedSummarizedExperiment
dim: 66 6
metadata(0):
assays(3): counts cpm logcounts
rownames(66): Gene4 Gene5 ... Gene197 Gene198
rowData names(3): feature_id gene_type Keep
colnames(6): Sample1 Sample2 ... Sample5 Sample6
colData names(4): SampleName Treatment Age Batch
```

And if we look at the resulting `rowData` we can see that it only contains the protein_coding features

```
rowData(coding)
```

```
DataFrame with 66 rows and 3 columns
         feature_id      gene_type      Keep
        <character>    <character> <logical>
Gene4         ID004 protein_coding      TRUE
Gene5         ID005 protein_coding      TRUE
Gene6         ID006 protein_coding      TRUE
```

```
Gene7         ID007 protein_coding      TRUE
Gene8         ID008 protein_coding      TRUE
...             ...            ...       ...
Gene190       ID190 protein_coding      TRUE
Gene191       ID191 protein_coding      TRUE
Gene192       ID192 protein_coding      TRUE
Gene197       ID197 protein_coding      TRUE
Gene198       ID198 protein_coding      TRUE
```

Each assay also reflects this operation

```
assay(coding, "cpm") |> head()
```

```
       Sample1  Sample2  Sample3  Sample4    Sample5   Sample6
Gene4 9565.261 1121.967 2067.540 9615.871  1402.4590 10436.152
Gene5 3869.930 9350.346 6661.908 4573.554 10275.6404  4092.315
Gene6 2384.180 2805.947 1062.807 6591.557   689.1799  9615.871
Gene7 1655.790 3473.709 3166.306 1217.493  4806.0909  7639.343
Gene8 1969.085 3239.030 5563.087 7384.070  9261.6023  8601.748
Gene9  856.531 2412.424 8425.051 4663.336  3467.8595  7840.655
```

Subsetting based on rowRanges

A closely related row-wise subsetting operation can be used if you have a `RangedSummarizedExperiment` (a `SummarizedExperiment` with a rowRanges slot) that allows you to perform operations on a `SummarizedExperiment` object like you would a GRanges object.

For example, let's say we only wanted to extract the features on Chromosome 2 only. Then we can use the `GenomicRanges` function `subsetByOverlaps` directly on our `SummarizedExperiment` object like so:

```
# Region of interest
roi <- GRanges(seqnames = "chr2", ranges = 1:1e7)

# Subset the SE object for only features on chr2
(chr2 <- subsetByOverlaps(se, roi))
```

```
class: RangedSummarizedExperiment
dim: 150 6
metadata(0):
assays(3): counts cpm logcounts
```

```
rownames(150): Gene51 Gene52 ... Gene199 Gene200
rowData names(3): feature_id gene_type Keep
colnames(6): Sample1 Sample2 ... Sample5 Sample6
colData names(4): SampleName Treatment Age Batch
```

You can see again that the `dim` changed reflecting our selection. Again, all of the associated assays and rowData have also been subsetted reflecting this change as well.

```
rowData(chr2)
```

```
DataFrame with 150 rows and 3 columns
         feature_id      gene_type      Keep
        <character>    <character> <logical>
Gene51        ID051 protein_coding      TRUE
Gene52        ID052         lncRNA     FALSE
Gene53        ID053 repeat_element     FALSE
Gene54        ID054 repeat_element     FALSE
Gene55        ID055 repeat_element     FALSE
...             ...            ...       ...
Gene196       ID196         lncRNA     FALSE
Gene197       ID197 protein_coding      TRUE
Gene198       ID198 protein_coding      TRUE
Gene199       ID199         lncRNA     FALSE
Gene200       ID200         lncRNA     FALSE
```

```
assay(chr2, "counts") |> head()
```

```
       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
Gene51     163      28     114      76     122      25
Gene52     220       4      83      98      23      43
Gene53      35      97      78     180      40      84
Gene54      43     213      18      82     158      81
Gene55     265      34      63      35     173     263
Gene56     131     106      91      40      18     131
```

```
rowRanges(chr2)
```

```
GRanges object with 150 ranges and 3 metadata columns:
          seqnames        ranges strand |  feature_id      gene_type      Keep
             <Rle>     <IRanges>  <Rle> | <character>    <character> <logical>
```

```
  Gene51     chr2 610565-610664      + |       ID051 protein_coding      TRUE
  Gene52     chr2 371337-371436      + |       ID052         lncRNA     FALSE
  Gene53     chr2 509774-509873      - |       ID053 repeat_element     FALSE
  Gene54     chr2 363826-363925      - |       ID054 repeat_element     FALSE
  Gene55     chr2 775744-775843      + |       ID055 repeat_element     FALSE
     ...      ...           ...    ... .         ...             ...       ...
 Gene196     chr2 828843-828942      + |       ID196         lncRNA     FALSE
 Gene197     chr2 725643-725742      - |       ID197 protein_coding      TRUE
 Gene198     chr2 580749-580848      + |       ID198 protein_coding      TRUE
 Gene199     chr2 404310-404409      + |       ID199         lncRNA     FALSE
 Gene200     chr2 303121-303220      - |       ID200         lncRNA     FALSE
 -------
 seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

There's also a few shortcuts on range operations using GRanges/SummarizedExperiments. See the help pages for `%over%`, `%within%`, and `%outside%`. For example:

```
all.equal(se[se %over% roi, ], subsetByOverlaps(se, roi))
```

```
[1] TRUE
```

Combining subsetting operations

Of course you don't have to perform one subsetting operation at a time. Like base R you can combine multiple expressions to subset a `SummarizedExperiment` object.

For example, to select only features labeled as repeat_elements and the Sample from 'Batch' A in the 'Control' group

```
(selected <- se[
  rowData(se)$gene_type == "repeat_element",
  se$Treatment == "Control" &
    se$Batch == "A"
])
```

```
class: RangedSummarizedExperiment
dim: 75 1
metadata(0):
assays(3): counts cpm logcounts
rownames(75): Gene1 Gene2 ... Gene186 Gene195
rowData names(3): feature_id gene_type Keep
colnames(1): Sample1
colData names(4): SampleName Treatment Age Batch
```

Saving a SummarizedExperiment

Since SummarizedExperiments keep basically all information about an experiment in one place, it is convenient to save the entire SummarizedExperiment object so that you can pick up an analysis where you left off or even to facilitate better sharing of data between collaborators.

You can save the entire SummarizedExperiment object with:

```
saveRDS(se, "/path/to/se.rds")
```

And when you want to read the same object back into R for your next analysis you can do so with:

```
se <- readRDS("/path/to/se.rds")
```

SummarizedExperiments in the real world

If you're working with any Bioconductor packages it's likely that the object you're working with either is a SummarizedExperiment or is inherited from one. For example, the DESeqDataSet from the DESeq2 package and BSseq objects from the bsseq package both inherit from a SummarizedExperiment and thus retain all of the same functionality as above. If you go to the SummarizedExperiment landing page and click "See More" under details you can see all of the packages that depend on SummarizedExperiment.

Also, many common methods are also implemented for SummarizedExperiment objects. For example, to simplify calculating counts-per-million above I could have simply used the edgeR::cpm() directly on the SummarizedExperiment object. Many functions in bioconductor packages know how to deal directly with SummarizedExperiments so you don't ever have to take the trouble extracting components and performing tedious calculations yourself.

```
assay(se, "cpm") <- edgeR::cpm(se)
```

I also left out any discussion of the metadata() slot of the SummarizedExperiment. The metadata slot is simply a list of any R object that contains information about the experiment. The metadata in the metadata slots are not subjected to the same subsetting rules as the other slots. In practice this assay contains additional information about the experiment as a whole. For example, I typically store bootstrap alignments for each sample here.

To add something to the SummarizedExperiment metadata slot you can do:

```
metadata(se)$additional_info <- "Experiment performed on 6 samples with three replicates eac
```

And to retrieve this:

```
metadata(se)$additional_info
```

```
[1] "Experiment performed on 6 samples with three replicates each"
```

Closing thoughts

Hopefully this was enough information to get you started using `SummarizedExperiments`. There's many things I left out such as different backings for storing out of memory data, a tidyverse interface to `SummarizedExperiment` objects, TreeSummarizedExperiments for microbiome data, MultiAssayExperiments for dealing with experiments containing multiomics data, and much more.

Please let me know your thoughts and if anything needs more clarification.

Differential Expression Analysis

This vignette outlines some common steps for RNA-seq analysis highlighting functions present in the `coriell` package. To illustrate some of the analysis steps I will borrow examples and data from the rnaseqGene and RNAseq123 Bioconductor workflows. Please check out the above workflows for more details regarding RNA-seq analysis.

This vignette contains my opinionated notes on performing RNA-seq analyses. I try to closely follow best practices from package authors but if any information is out of date or incorrect, please let me know.

Overview

Differential gene expression analysis using RNA-seq typically consists of several steps:

1. Quality control of the fastq files with a tool like FastQC or fastp
2. Alignment of fastq files to a reference genome using a splice-aware aligner like STAR *or* transcript quantification using a pseudoaligner like Salmon.
3. If using a genome aligner, read counting with Rsubread::featureCounts or HTSeq count to generate gene counts. If using a transcript aligner, importing gene-level counts using the appropriate offsets with tximport or tximeta
4. Quality control plots of the count level data including PCA, heatmaps, relative-log expression boxplots, density plots of count distributions, and parallel coordinate plots of libraries. Additionally, check the assumptions of global scaling normalization.
5. Differential expression testing on the raw counts using edgeR, DESeq2, baySeq, or limma::voom
6. Creation of results plots such as volcano or MA plots.
7. Gene ontology analysis of interesting genes.
8. Gene set enrichment analysis.

Quality Control

fastp has quickly become my favorite tool for QC'ing fastq files primarily because it is fast and produces nice looking output files that are also amenable to summarization with MultiQC. fastp can also perform adapter trimming on paired-end reads. I tend to fall in the camp that

believes read quality trimming is not necessary for RNA-seq alignment. However, I have never experienced *worse* results after using the adapter trimming with fastp so I leave it be and just inspect the output carefully.

A simple bash script for running fastp over a set of fastq files might look something like this:

```
#!/usr/bin/env bash
#
# Run fastp on the raw fastq files
#
# ------------------------------------------------------------------------
set -Eeou pipefail

FQ=/path/to/00_fastq       # Directory containing raw fastq files
SAMPLES=sample-names.txt   # A text file listing basenames of fastq files
OUT=/path/to/put/01_fastp  # Where to save the fastp results
THREADS=8

mkdir -p $OUT

for SAMPLE in $(cat $SAMPLES)
do
    fastp -i $FQ/${SAMPLE}_R1.fq.gz \
          -I $FQ/${SAMPLE}_R2.fq.gz \
          -o $OUT/${SAMPLE}.trimmed.1.fq.gz \
          -O $OUT/${SAMPLE}.trimmed.2.fq.gz \
          -h $OUT/${SAMPLE}.fastp.html \
          -j $OUT/${SAMPLE}.fastp.json \
          --detect_adapter_for_pe \
          -w $THREADS
done
```

Where `sample-names.txt` is a simple text file with each basename like so:

```
Control1
Control2
Control3
Treatment1
Treatment2
Treatment3
```

It is important to name the results files with `*.fastp.{json|html}` so that `multiqc` can recognize the extensions and combine the results automatically.

Alignment and Quantification

Salmon

I tend to perform quantification with Salmon to obtain transcript-level counts for each sample. A simple bash script for performing quantification with Salmon looks like:

```
#!/usr/bin/env bash
#
# Perform transcript quantification with Salmon
#
# ------------------------------------------------------------------------------
set -Eeou pipefail

SAMPLES=sample-names.txt  # Same sample-names.txt file as above
IDX=/path/to/salmon-idx   # Index used by Salmon
FQ=/path/to/01_fastp      # Directory containing the fastp output
OUT=/path/to/02_quants    # Where to save the Salmon results
THREADS=12

mkdir -p $OUT

for SAMPLE in $(cat $SAMPLES)
do
    salmon quant \
           -i $IDX \
           -l A \
           -1 $FQ/${SAMPLE}.trimmed.1.fq.gz \
           -2 $FQ/${SAMPLE}.trimmed.2.fq.gz \
           --gcBias \
           --seqBias \
           --threads $THREADS \
           -o $OUT/${SAMPLE}_quants
done
```

I tend to always use the `--gcBias` and `--seqBias` flags as they don't impair accuracy in the absence of biases (quantification just takes a little longer).

STAR

Sometimes I also need to produce genomic coordinates for alignments. For this purpose I tend to use STAR to generate BAM files as well as produce gene-level counts with it's inbuilt

HTSeq-count functionality. A simple bash script for running STAR might look like:

```
#!/usr/bin/env bash
#
# Align reads with STAR
#
# -----------------------------------------------------------------------------
set -Eeou pipefail

SAMPLES=sample-names.txt   # Same sample-names.txt file as above
FQ=/path/to/01_fastp       # Directory containing the fastp output
OUT=/path/to/03_STAR_outs  # Where to save the STAR results
IDX=/path/to/STAR-idx      # Index used by STAR for alignment
THREADS=24

mkdir -p $OUT

for SAMPLE in $(cat $SAMPLES)
do
  STAR --runThreadN $THREADS \
       --genomeDir $IDX \
       --readFilesIn ${FQ}/${SAMPLE}.trimmed.1.fq.gz ${FQ}/${SAMPLE}.trimmed.2.fq.gz \
       --readFilesCommand zcat \
       --outFilterType BySJout \
       --outFileNamePrefix ${OUT}/${SAMPLE}_ \
       --alignSJoverhangMin 8 \
       --alignSJDBoverhangMin 1 \
       --outFilterMismatchNmax 999 \
       --outFilterMismatchNoverReadLmax 0.04 \
       --alignIntronMin 20 \
       --alignIntronMax 1000000 \
       --alignMatesGapMax 1000000 \
       --outMultimapperOrder Random \
       --outSAMtype BAM SortedByCoordinate \
       --quantMode GeneCounts;
done
```

For STAR I tend to use the ENCODE default parameters above for human samples and also output gene level counts using the `--quantMode GeneCounts` flag.

Generating a matrix of gene counts

The recommended methods for performing differential expression analysis implemented in edgeR, DESeq2, baySeq, and limma::voom all require raw count matrices as input data.

Importing transcript-level counts from `Salmon`

We use `R` to import the quant files into the active session. `tximeta` will download the appropriate metadata for the reference genome used and import the results as a `SummarizedExperiment` object. Check out the tutorial for working with `SummarizedExperiment` objects if you are unfamiliar with their structure.

The code below will create a data.frame mapping sample names to file paths containing quantification results. This data.frame is then used by `tximeta` to import `Salmon` quantification results at the transcript level (along with transcript annotations). Then, we use `summarizeToGene()` to summarize the tx counts to the gene level. Finally, we transform the `SummarizedExperiment` object to a `DGEList` for use in downstream analysis with `edgeR`

```
library(tximeta)
library(edgeR)


quant_files <- list.files(
  path = "02_quants",
  pattern = "quant.sf",
  full.names = TRUE,
  recursive = TRUE
)

# Extract samples names from filepaths
names(quant_files) <- gsub("02_quants", "", quant_files, fixed = TRUE)
names(quant_files) <- gsub("_quants/quant.sf", "", names(quant_files), fixed = TRUE)

# Create metadata for import
coldata <- data.frame(
  names = names(quant_files),
  files = quant_files,
  group = factor(rep(c("Control", "Treatment"), each = 3))
  )

# Import transcript counts with tximeta
se <- tximeta(coldata)
```

```
# Summarize tx counts to the gene-level
gse <- summarizeToGene(se, countsFromAbundance = "scaledTPM")

# Import into edgeR for downstream analysis
y <- SE2DGEList(gse)
```

Importing gene counts from STAR

If you used STAR to generate counts with HTSeq-count then `edgeR` can directly import the results for downstream analysis like so:

```
library(edgeR)


# Specify the filepaths to gene counts from STAR
count_files <- list.files(
  path = "03_STAR_outs",
  pattern = "*.ReadsPerGene.out.tab",
  full.names = TRUE
  )

# Name the file with their sample names
names(count_files) <- gsub(".ReadsPerGene.out.tab", "", basename(count_files))

# Import HTSeq counts into a DGEList
y <- readDGE(
  files = count_files,
  columns = c(1, 2),  # Gene name and 'unstranded' count columns
  group = factor(rep(c("Control", "Treatment"), each = 3)),
  labels = names(count_files)
  )
```

Counting reads with `featureCounts()`

Another (preferred) option for generating counts from BAM files is to use the function `featureCounts()` from the `Rsubread` R package. `featureCounts()` is very fast and has many options to control exactly how reads are counted. `featureCounts()` is also very general. For example, you can use `featureCounts()` to count reads over exons, introns, or arbitrary genomic ranges - it's a very useful tool.

`featureCounts()` can use an inbuilt annotation or take a user supplied GTF file to count reads over. The inbuilt annotation (NCBI RefSeq) works particularly well with downstream functions implemented in `edgeR`. See `?Rsubread::featureCounts()` for more information about using your own GTF file.

The resulting object can also be easily coerced to a `DGEList` for downstream analysis.

```
library(Rsubread)


# Specify the path to the BAM files produced by STAR
bam_files <- list.files(
  path = "path/to/bam_files",
  pattern = "*.bam$",
  full.names = TRUE
  )

# Optionally name the bam files
names(bam_files) <- gsub(".bam", "", basename(bam_files))

# Count reads over genes for a paired-end library
fc <- featureCounts(
  files = bam_files,
  annot.inbuilt = "hg38",
  isPairedEnd = TRUE,
  nThreads = 12
)

# Coerce to DGEList for downstream analysis
y <- edgeR::featureCounts2DGEList(fc)
```

Test data

We will use data from the airway package to illustrate differential expression analysis steps. Please see Section 2 of the *rnaseqGene* workflow for more information.

Below, we load the data from the airway package and use `SE2DGEList` to convert the object to a `DGElist` for use with `edgeR`.

```
library(airway)
library(edgeR)
```

```
# Load the SummarizedExperiment object
data(airway)

# Set the group levels
airway$group <- factor(airway$dex, levels = c("untrt", "trt"))

# Convert to a DGEList to be consistent with above steps
y <- SE2DGEList(airway)
```

Library QC

Before we perform differential expression analysis it is important to explore the samples' library distributions to ensure good quality before downstream analysis. There are several diagnostic plots we can use for this purpose implemented in the `coriell` package. However, first we must remove any features that have too low of counts for meaningful differential expression analysis. This can be achieved using `edgeR::filterByExpr()`.

```
# Determine which genes have enough counts to keep around
keep <- filterByExpr(y)

# Remove the unexpressed genes
y <- y[keep,,keep.lib.sizes = FALSE]
```

At this stage it is often wise to perform library QC on the library size normalized counts. This will give us an idea about potential global expression differences and potential outliers *before* introducing normalization factors. We can use `edgeR` to generate log2 counts-per-million values for the retained genes.

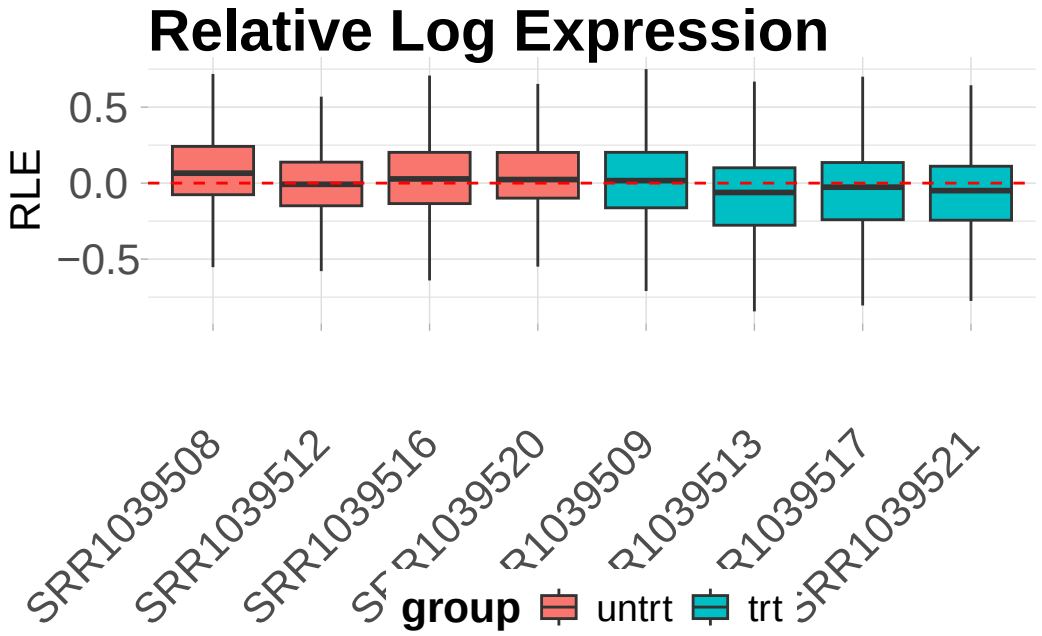
```
logcounts <- cpm(y, log = TRUE)
```

Relative log-expression boxplots

The first diagnostic plot we can look at is a plot of the relative log expression values. RLE plots are good diagnostic tools for evaluating unwanted variation in libraries.

```
library(ggplot2)
library(coriell)
```

```
plot_boxplot(logcounts, metadata = y$samples, fillBy = "group",
             rle = TRUE, outliers = FALSE) +
  labs(title = "Relative Log Expression",
       x = NULL,
       y = "RLE",
       color = "Treatment Group")
```

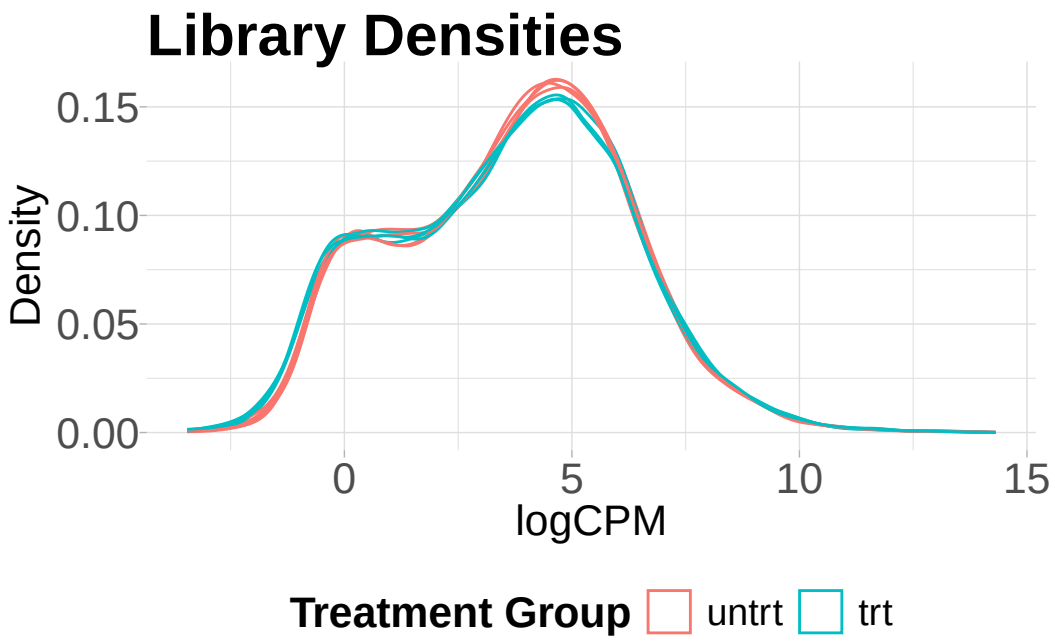


We can see from the above RLE plot that the samples are centered around zero and have mostly similar distributions. It is also clear that two of the samples, "SRR1039520" and "SRR1039521", have slightly different distributions than the others.

Library density plots

Library density plots show the density of reads corresponding to a particular magnitude of counts. Shifts of these curves should align with group differences and generally samples from the same group should have overlapping density curves

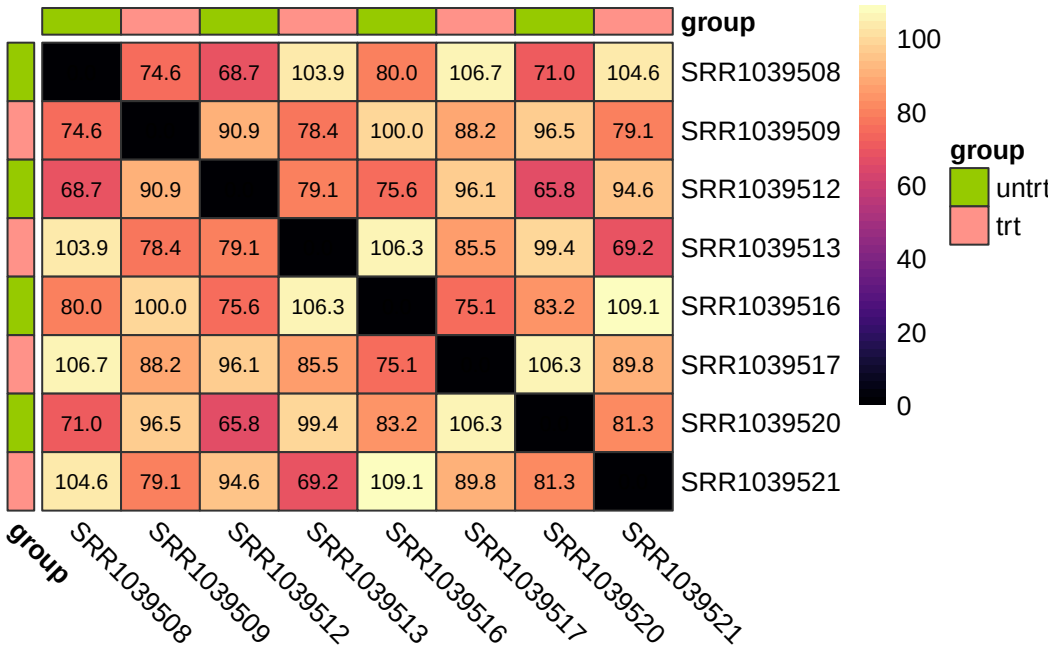
```
plot_density(logcounts, metadata = y$samples, colBy = "group") +
  labs(title = "Library Densities",
       x = "logCPM",
       y = "Density",
       color = "Treatment Group")
```



Sample vs Sample Distances

We can also calculate the euclidean distance between all pairs of samples and display this on a heatmap. Again, samples from the same group should show smaller distances than sample pairs from differing groups.

```
plot_dist(logcounts, metadata = y$samples[, "group", drop = FALSE])
```

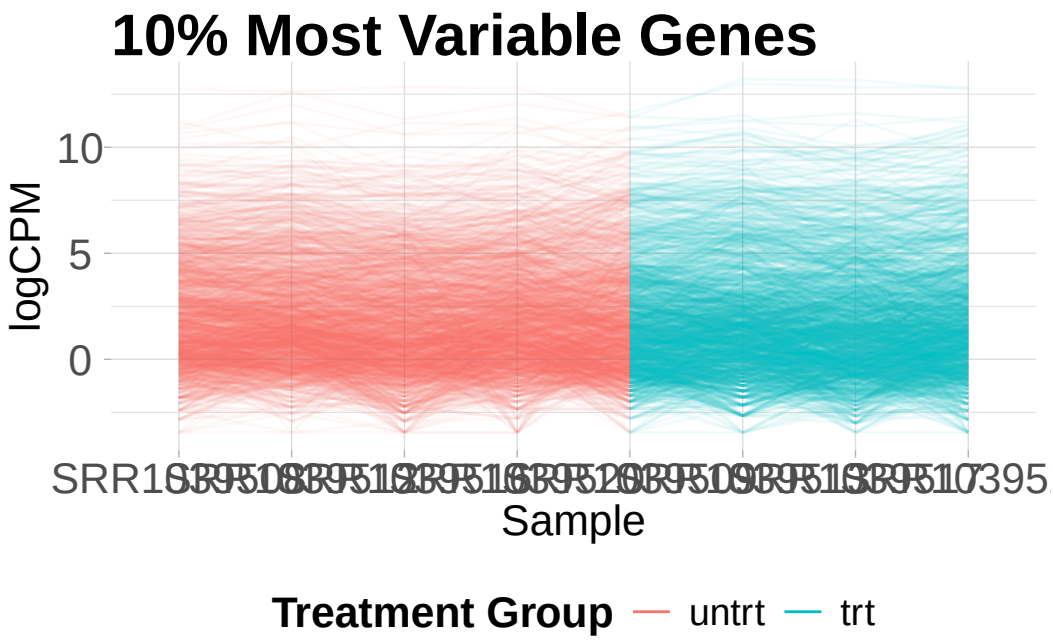


Parallel coordinates plot

Parallel coordinates plots are useful for giving you an idea of how the most variable genes change between treatment groups. These plots show the expression of each gene as a line on the y-axis traced between samples on the x-axis.

```
plot_parallel(logcounts, y$samples, colBy = "group",
              removeVar = 0.9, alpha = 0.05) +
  labs(title = "10% Most Variable Genes",
       x = "Sample",
       y = "logCPM",
       color = "Treatment Group")
```

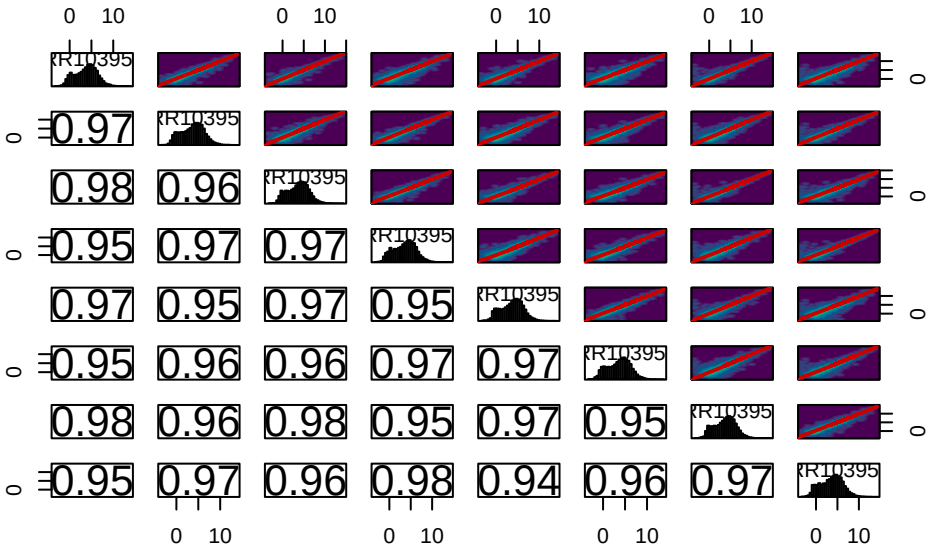
```
Removing 90% lowest variance features...
```



Correlations between samples

We can also plot the pairwise correlations between all samples. These plots can be useful for identifying technical replicates that deviate from the group

```
plot_cor_pairs(logcounts, cex_labels = 1)
```



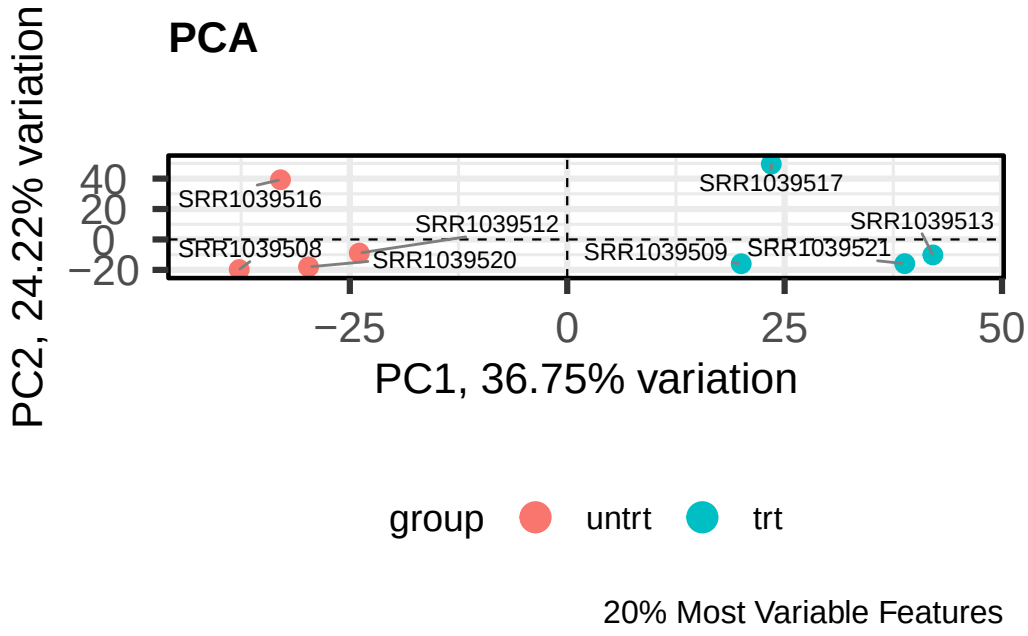
PCA

Principal components analysis is an unsupervised method for reducing the dimensionality of a dataset while maintaining its fundamental structure. PCA biplots can be used to examine sample groupings following PCA. These biplots can reveal overall patterns of expression as well as potential problematic samples prior to downstream analysis. For simple analyses we expect to see the 'main' effect primarily along the first component.

I like to use the PCAtools package for quickly computing and plotting principal components. For more complicated experiments I have also found UMAP (see `coriell::UMAP()`) to be useful for dimensionality reduction (although using UMAP is not without its problems for biologists).

```
library(PCAtools)


# Perform PCA on the 20% most variable genes
# Center and scale the variable after selecting most variable
pca_result <- pca(
  logcounts,
  metadata = y$samples,
  center = TRUE,
```

```
  scale = TRUE,
  removeVar = 0.8
  )

# Show the PCA biplot
biplot(
  pca_result,
  colby = "group",
  hline = 0,
  vline = 0,
  hlineType = 2,
  vlineType = 2,
  legendPosition = "bottom",
  title = "PCA",
  caption = "20% Most Variable Features"
  )
```



Assessing global scaling normalization assumptions

Most downstream differential expression testing methods apply a global scaling normalization factor to each library prior to DE testing. Applying these normalization factors when there are global expression differences can lead to spurious results. In typical experiments this is

usually not a problem but when dealing with cancer or epigenetic drug treatment this can actually lead to many problems if not identified.

To identify potential violations of global scaling normalization I use the quantro R package. `quantro` uses two data driven approaches to assess the appropriateness of global scaling normalization. The first involves testing if the medians of the distributions differ between groups. These differences could indicate technical or real biological variation. The second test assesses the ratio of between group variability to within group variability using a permutation test similar to an ANOVA. If this value is large, it suggests global adjustment methods might not be appropriate.

```
library(quantro)


# Initialize multiple (8) cores for permutation testing
doParallel::registerDoParallel(cores = 8)

# Compute the qstat on the filtered libraries
qtest <- quantro(y$counts, groupFactor = y$samples$group, B = 500)
```

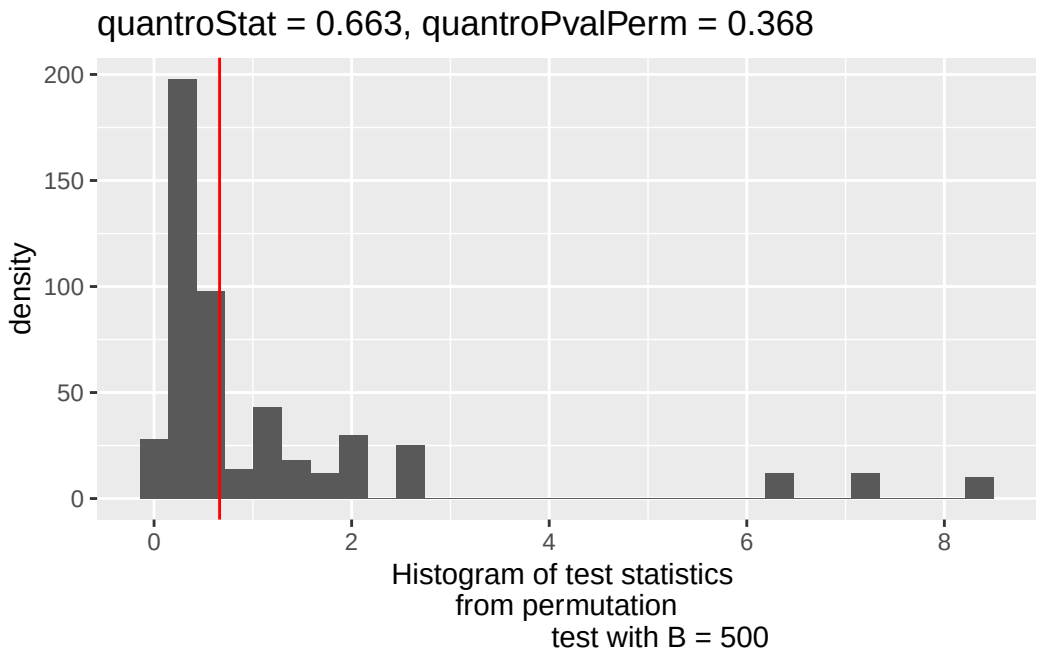
Now we can assess the results. We can use `anova()` to test for differences in medians across groups. Here, they do not significantly differ.

```
anova(qtest)
```

```
Analysis of Variance Table

Response: objectMedians
            Df  Sum Sq Mean Sq F value Pr(>F)
groupFactor  1  1984.5  1984.5  0.3813 0.5596
Residuals    6 31225.5  5204.3
```

We can also plot the results of the permutation test to see the between:within group ratios. Again, there are no large differences in this dataset suggesting that global scaling normalization such as TMM is appropriate.

```
quantroPlot(qtest)
```



Differential expression testing with `edgeR`

After removing lowly expressed features and checking the assumptions of normalization we can perform downstream differential expression testing with `edgeR`. The edgeR manual contains a detailed explanation of all steps involved in differential expression testing.

In short, we need to specify the experimental design, estimate normalization factors, fit the models, and perform DE testing.

Creating the experimental design

Maybe the most important step in DE analysis is properly constructing a design matrix. The details of design matrices are outside of the scope of this tutorial but a good overview can be found here. Generally, your samples will fall nicely into several well defined groups, facilitating the use of a design matrix without an intercept e.g. `design ~ model.matrix(~0 + group, ...)`. This kind of design matrix makes it relatively simple to construct contrasts that describe exactly what pairs of groups you want to compare.

Since this example experiment is simply comparing treatments to control samples we can model the differences in means by using a model *with* an intercept where the intercept is the mean of the control samples and the 2nd coefficient represents the differences in the treatment group.

```
# Model with intercept
design <- model.matrix(~group, data = y$samples)
```

We can make an equivalent model and test *without* an intercept like so:

```
# A means model
design_no_intercept <- model.matrix(~0 + group, data = y$samples)

# Construct contrasts to test the difference in means between the groups
cm <- makeContrasts(
  Treatment_vs_Control = grouptrt - groupuntrt,
  levels = design_no_intercept
)
```

The choice of which design is up to you. I typically use whatever is clearer for the experiment at hand. In this case, that is the model with an intercept.

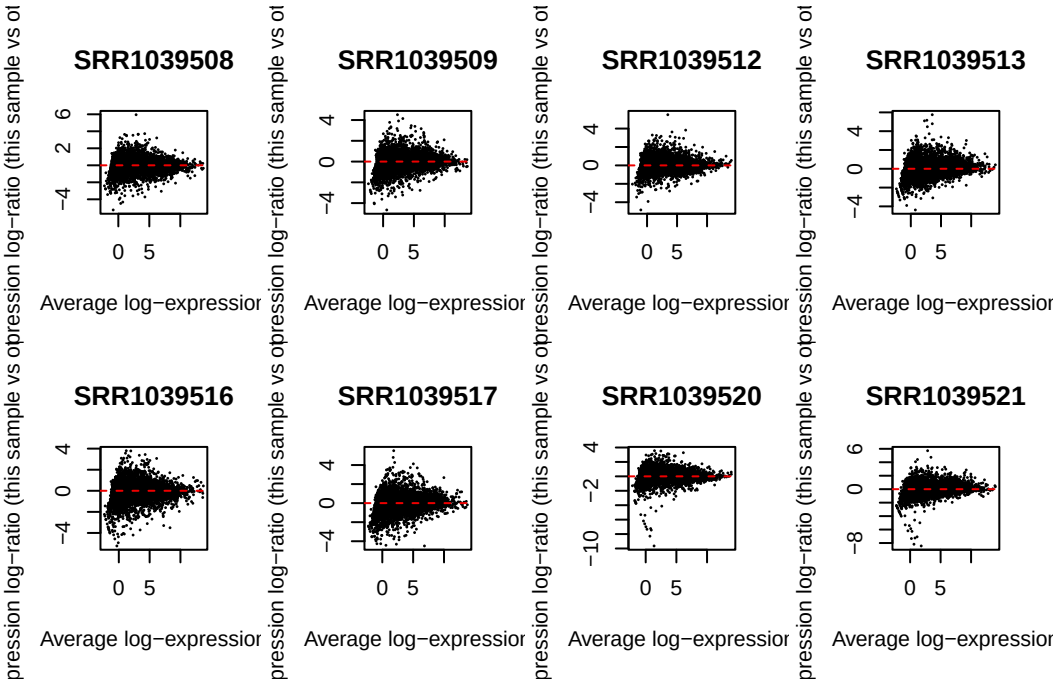
Estimating normalization factors

We use `edgeR` to calculate trimmed mean of the M-value (TMM) normalization factors for each library.

```
# Estimate TMM normalization factors
y <- normLibSizes(y)
```

We can check the normalization by creating MA plots for each library. The bulk of the data should be centered on zero without any obvious differences in the logFC as a function of average abundance.

```
par(mfrow = c(2, 4))
for (i in 1:ncol(y)) {
  plotMD(cpm(y, log = TRUE), column = i)
  abline(h = 0, lty = 2, col = "red2")
}
```



What to do if global scaling normalization is violated?

Above I described testing for violations of global scaling normalization. So what should we do if these assumptions are violated and we don't have a good set of control genes or spike-ins etc.?

If we believe that the differences we are observing are due to true biological phenomena (**this is a big assumption**) then we can try to apply a method such as smooth quantile normalization to the data using the qsmooth package.

Below I will show how to apply `qsmooth` to our filtered counts and then calculate offsets to be used in downstream DE analysis with `edgeR`. Please note **this is not a benchmarked or 'official' workflow** just a method that I have implemented based on reading forums and github issues.

```
library(qsmooth)


# Compute the smooth quantile factors
qs <- qsmooth(y$counts, group_factor = y$samples$group)

# Extract the qsmooth transformed data
qsd <- qsmoothData(qs)
```

```
# Calculate offsets to be used by edgeR in place of norm.factors
# Offsets are on the natural log scale. Add a small offset to avoid
# taking logs of zero
offset <- log(y$counts + 0.1) - log(qsd + 0.1)

# Scale the offsets for internal usage by the DGEList object
# Now the object is ready for downstream analysis
y <- scaleOffset(y, offset = offset)

# To create logCPM values with the new norm factors use
lcpm <- cpm(y, offset = y$offset, log = TRUE)
```

Fit the model

New in edgeR 4.0 is the ability to estimate dispersions while performing the model fitting step. I typically tend to 'robustify' the fit to outliers. Below I will perform dispersion estimation in legacy mode so that we can use competitive gene set testing later. If we want to use the new workflow we can use the following:

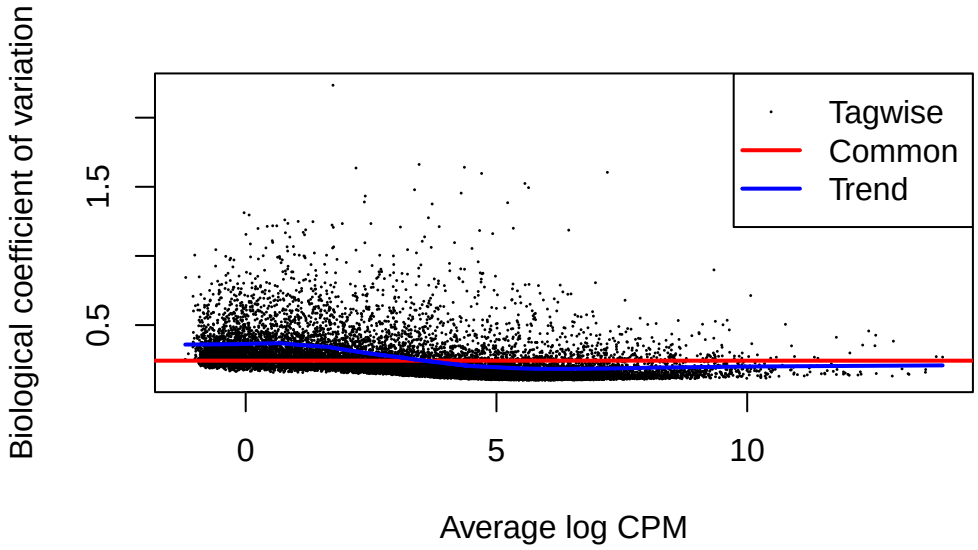
```
# edgeR 4.0 workflow
fit <- glmQLFit(y, design, legacy = FALSE, robust = TRUE)
```

We will continue with the legacy workflow.

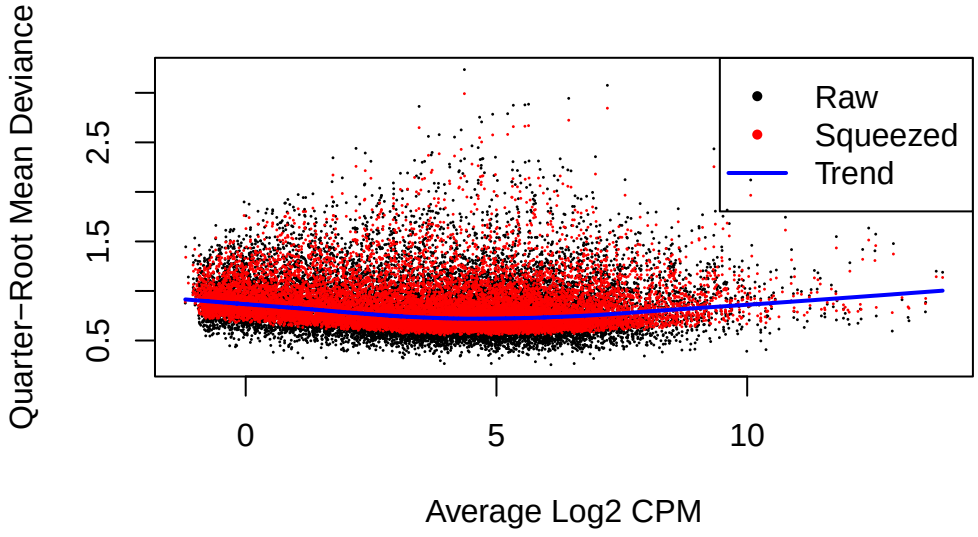
```
y <- estimateDisp(y, design, robust = TRUE)
fit <- glmQLFit(y, design, robust = TRUE, legacy = TRUE)
```

It's always a good idea at this step to check some of the diagnostic plots from `edgeR`

```
# Show the biological coefficient of variation
plotBCV(y)
```




```
# Show the dispersion estimates
plotQLDisp(fit)
```



Test for differential expression

Now that the models have been fit we can test for differential expression.

```
# Test the treatment vs control condition
qlf <- glmQLFTest(fit, coef = 2)
```

Often it is more biologically relevant to give more weight to higher fold changes. This can be achieved using `glmTreat()`. **NOTE** do not use `glmQLFTest()` and then filter by fold-change - you destroy the FDR correction!

When testing against a fold-change we can use relatively modest values since the fold-change must exceed this threshold before being considered for significance. Values such as `log2(1.2)` or `log2(1.5)` work well in practice.

```
trt_vs_control_fc <- glmTreat(fit, coef = 2, lfc = log2(1.2))
```

In any case, the results of the differential expression test can be extracted to a data.frame for downstream plotting with `coriell::edger_to_df()`. This function simply returns a data.frame of all results from the differential expression object in the same order as `y`. (i.e. `topTags(..., n=Inf, sort.by="none")`)

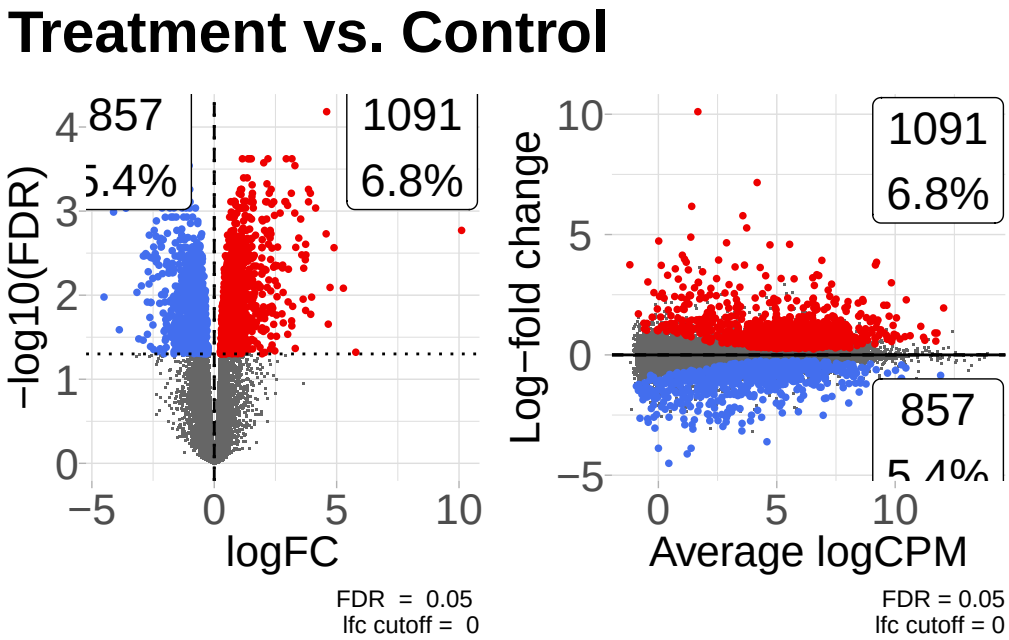
```
de_result <- edger_to_df(qlf)
```

Plotting DE results

The two most common plots for differential expression analysis results are the volcano plot and the MA plot. Volcano plots display the negative log10 of the significance value on the y-axis vs the log2 fold-change on the x-axis. MA plots show the average expression of the gene on the x-axis vs the log2 fold-change of the gene on the y-axis. The `coriell` package includes functions for producing both.

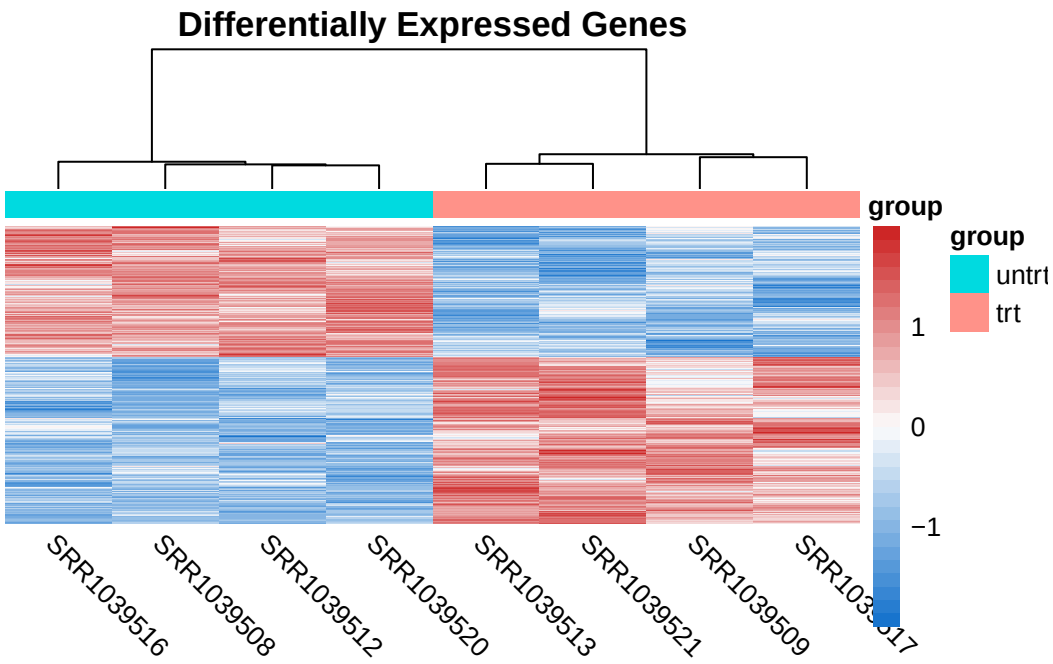
```
library(patchwork)


# Create a volcano plot of the results
v <- plot_volcano(de_result, fdr = 0.05)

# Create and MA plot of the results
m <- plot_md(de_result, fdr = 0.05)
```

```
# Patch both plots together
(v | m) +
  plot_annotation(title = "Treatment vs. Control") &
  theme_coriell()
```



The `coriell` package also has a function for quickly producing heatmaps with nice defaults for RNA-seq. Sometimes it's useful to show the heatmaps of the DE genes.

```
# Compute logCPM values after normalization
lcpm <- cpm(y, log = TRUE)

# Determine which of the genes in the result were differentially expressed
is_de <- de_result$FDR < 0.05

# Produce a heatmap from the DE genes
quickmap(
  x = lcpm[is_de, ],
  metadata = y$samples[, "group", drop = FALSE],
  main = "Differentially Expressed Genes"
  )
```



Competitive gene set testing with `camera()`

I've recently become aware of some of the problems with gene set enrichment analysis using the `fgsea` package. Following Gordon Smyth's advice, I have switched all of my pipelines to using competitive gene set testing (when appropriate) in `limma` to avoid problems with correlated genes.

Below we use the `msigdbr` R package to retrieve HALLMARK gene sets and then use `edgeR::camera()` for gene set testing.

```
library(msigdbr)


# Get the HALLMARK gene set data
msigdb_hs <- msigdbr(species = "Homo sapiens", category = "H")
```

```
Warning: The `category` argument of `msigdbr()` is deprecated as of msigdbr 10.0.0.
i Please use the `collection` argument instead.
```

```
# Split into list of gene names per HALLMARK pathway
msigdb_hs <- split(as.character(msigdb_hs$gene_symbol), msigdb_hs$gs_name)

# Convert the gene sets into lists of indeces for edgeR
idx <- ids2indices(gene.sets = msigdb_hs, identifiers = y$genes$gene_name)
```

Perform gene set testing. Note here we can use `camera()` `mroast()`, or `romer()` depending on the hypothesis being tested. The above setup code provides valid input for all of the above functions.

See this comment from Aaron Lun describing the difference between `camera()` and `roast()`. For GSEA like hypothesis we can use `romer()`

> `roast()` performs a self-contained gene set test, where it looks for any DE within the set of genes. `camera()` performs a competitive gene set test, where it compares the DE within the gene set to the DE outside of the gene set.

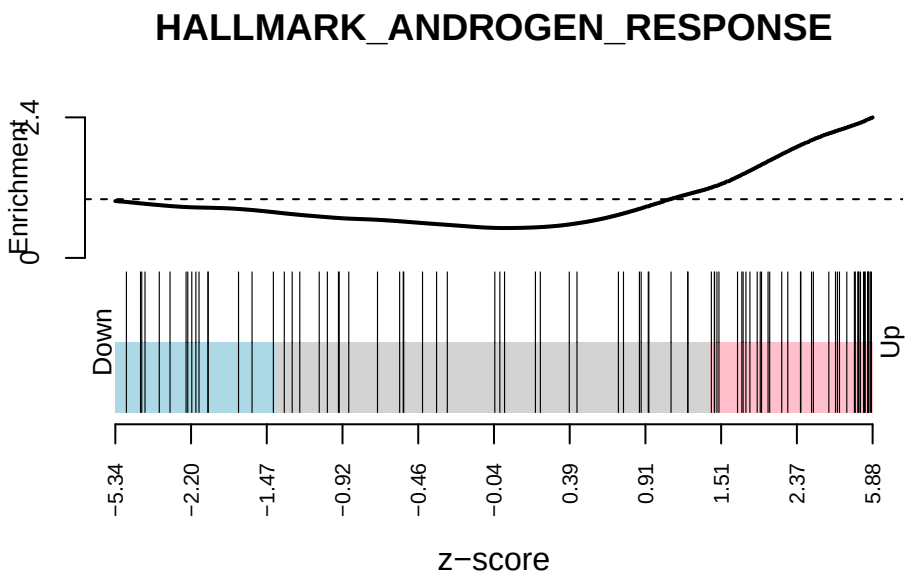
```
# Use camera to perform competitive gene set testing
camera_result <- camera(y, idx, design, contrast = 2)

# Use mroast for rotational gene set testing - bump up number of rotations
mroast_result <- mroast(y, idx, design, contrast = 2, nrot = 1e4)

# Use romer for GSEA like hypothesis testing
romer_result <- romer(y, idx, design, contrast = 2)
```

We can also perform a pre-ranked version of the camera test using `cameraPR()`. To use the pre-ranked version we need to create a ranking statistic. The suggestion from Gordon Smyth is to derive a z-statistic from the F-scores like so:

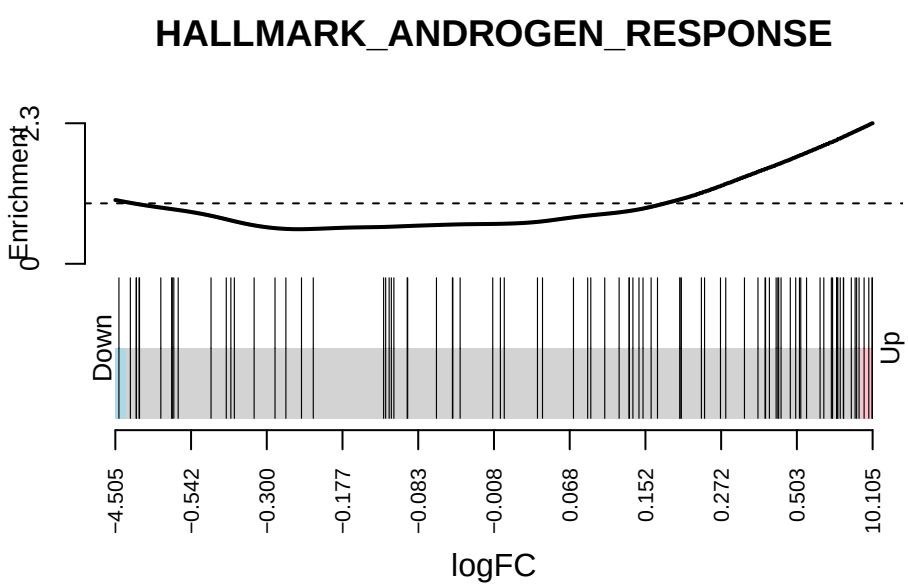
```
t_stat <- sign(de_result$logFC) * sqrt(de_result$`F`)
z <- zscoreT(t_stat, df = qlf$df.total)

# Name the stat vector with the gene names
names(z) <- de_result$gene_name

# Use the z-scores as the ranking stat for cameraPR
camera_pr_result <- cameraPR(z, idx)
```

Another useful plot to show following gene set testing is a barcodeplot. We The barcodeplot displays the enrichment of a given signature for a ranked list of genes. The `limma::barcodeplot()` function allows us to easily create these plots for any of the gene sets of interest using any ranking stat of our choice.

```r
# Show barcodeplot using the z-scores
barcodeplot(
  z,
  index = idx[["HALLMARK_ANDROGEN_RESPONSE"]],
  main = "HALLMARK_ANDROGEN_RESPONSE",
  xlab = "z-score"
  )
```




```r
# Or you can use the logFC
barcodeplot(
  de_result$logFC,
  index = idx[["HALLMARK_ANDROGEN_RESPONSE"]],
  main = "HALLMARK_ANDROGEN_RESPONSE",
  xlab = "logFC"
  )
```



Gene ontology (GO) over-representation test

Over-representation analysis can be performed with the `clusterProfiler` package. Here, instead of using the entire gene list as input we select separate sets of up and down-regulated genes and test to see if these sets are enriched in our differentially expressed gene list.

```
library(clusterProfiler)
library(org.Hs.eg.db)


# Split the genes into up and down
up_genes <- subset(
  de_result,
  FDR < 0.05 & logFC > 0,
  "gene_name",
  drop = TRUE
  )

down_genes <- subset(
  de_result,
  FDR < 0.05 & logFC < 0,
```

```
  "gene_name",
  drop = TRUE
  )

# Extract the list of all genes expressed in the experiment
# to use as a background set
universe <- unique(y$genes$gene_name)
```

Create results objects for each set of genes

```
ego_up <- enrichGO(
  gene = up_genes,
  universe = universe,
  OrgDb = org.Hs.eg.db,
  keyType = "SYMBOL",
  ont = "ALL",
  pAdjustMethod = "BH",
  pvalueCutoff = 0.01,
  qvalueCutoff = 0.05,
  readable = TRUE
  )

ego_down <- enrichGO(
  gene = down_genes,
  universe = universe,
  OrgDb = org.Hs.eg.db,
  keyType = "SYMBOL",
  ont = "ALL",
  pAdjustMethod = "BH",
  pvalueCutoff = 0.01,
  qvalueCutoff = 0.05,
  readable = TRUE
  )
```

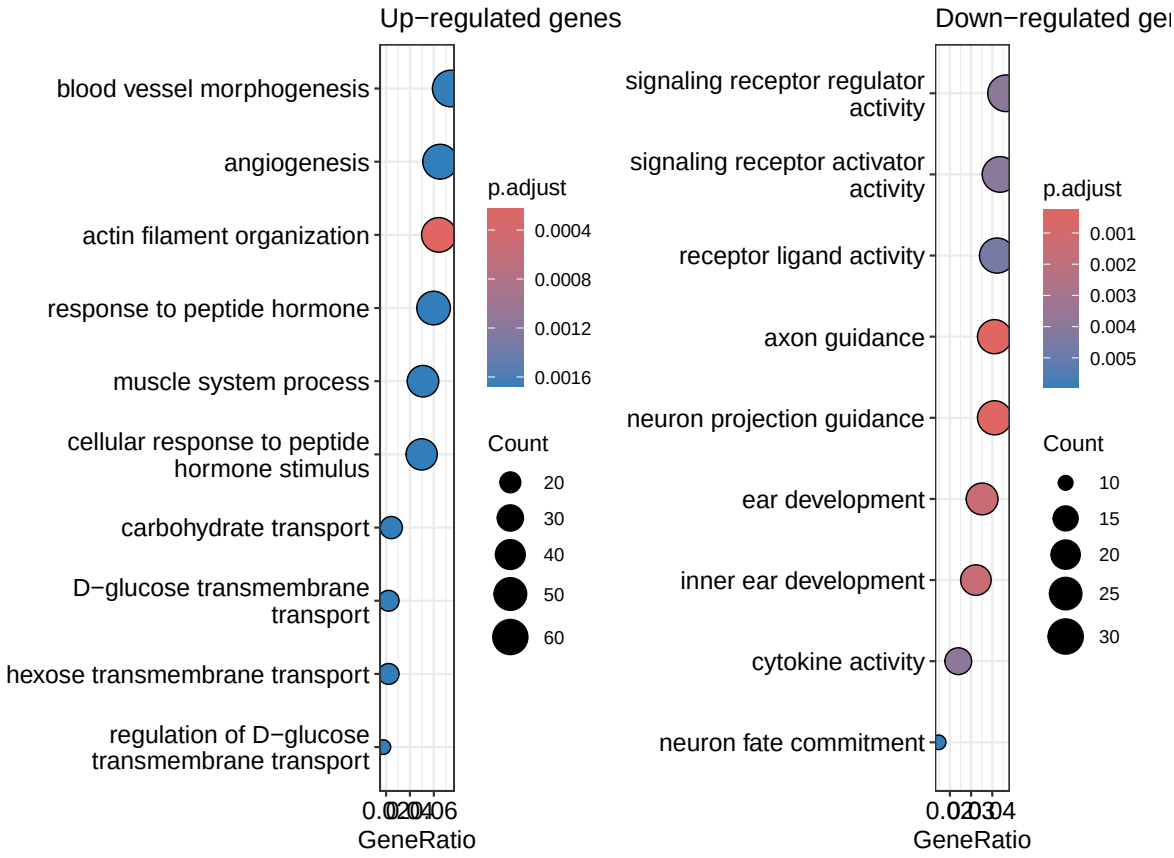
These results can be converted to data.frames and combined with:

```
ego_up_df <- data.frame(ego_up)
ego_down_df <- data.frame(ego_down)

ego_df <- data.table::rbindlist(
  list(up = ego_up_df, down = ego_down_df),
```

```
  idcol = "Direction"
  )
```

Or the results can be plotted as dotplots with:

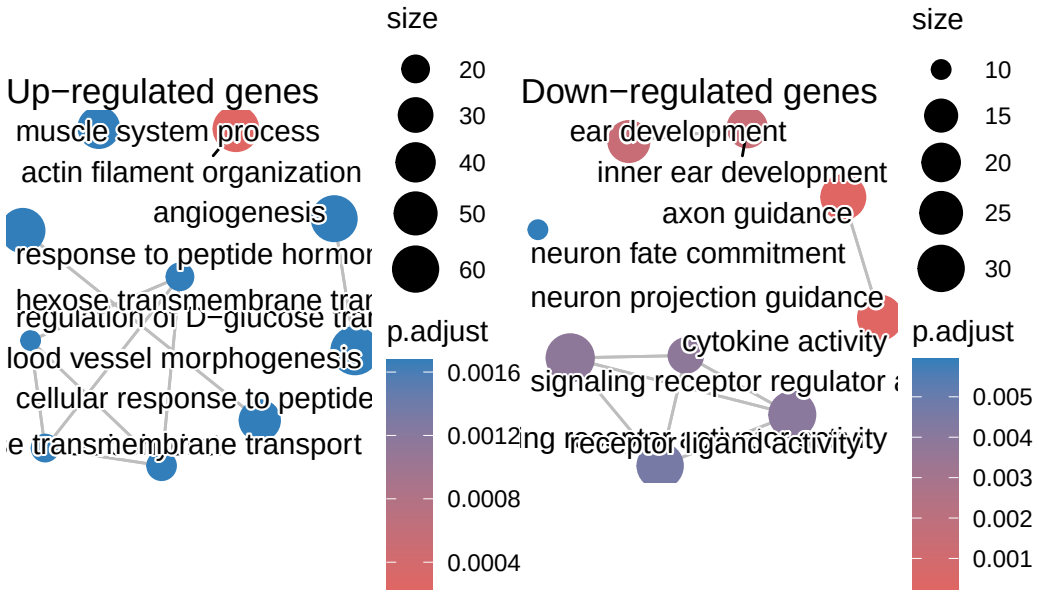
```
d1 <- dotplot(ego_up) + labs(title = "Up-regulated genes")
d2 <- dotplot(ego_down) + labs(title = "Down-regulated genes")

d1 | d2
```



You can also create a nice enrichment map showing similarity between the significant GO terms like so:

```
em_up <- enrichplot::pairwise_termsim(ego_up)
em_down <- enrichplot::pairwise_termsim(ego_down)
```

```
p1 <- enrichplot::emapplot(em_up, showCategory = 10, min_edge = 0.5) +
  labs(title = "Up-regulated genes")
p2 <- enrichplot::emapplot(em_down, showCategory = 10, min_edge = 0.5) +
  labs(title = "Down-regulated genes")

p1 | p2
```



Session Info

```
sessionInfo()
```

```
R version 4.5.1 (2025-06-13)
Platform: x86_64-pc-linux-gnu
Running under: Pop!_OS 22.04 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so;  LAPACK version 3
```

```
locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: America/New_York
tzcode source: system (glibc)

attached base packages:
[1] stats4    stats     graphics  grDevices datasets  utils     methods
[8] base

other attached packages:
 [1] org.Hs.eg.db_3.21.0         AnnotationDbi_1.70.0
 [3] clusterProfiler_4.16.0      msigdbr_25.1.1
 [5] patchwork_1.3.1             quantro_1.40.0
 [7] PCAtools_2.20.0             ggrepel_0.9.6
 [9] coriell_0.17.0              ggplot2_3.5.2
[11] edgeR_4.6.3                 limma_3.64.1
[13] airway_1.28.0               SummarizedExperiment_1.38.1
[15] Biobase_2.68.0              GenomicRanges_1.60.0
[17] GenomeInfoDb_1.44.1         IRanges_2.42.0
[19] S4Vectors_0.46.0            BiocGenerics_0.54.0
[21] generics_0.1.4              MatrixGenerics_1.20.0
[23] matrixStats_1.5.0

loaded via a namespace (and not attached):
  [1] splines_4.5.1             BiocIO_1.18.0
  [3] ggplotify_0.1.2           bitops_1.0-9
  [5] R.oo_1.27.1               tibble_3.3.0
  [7] preprocessCore_1.70.0     XML_3.99-0.18
  [9] lifecycle_1.0.4           doParallel_1.0.17
 [11] lattice_0.22-7            MASS_7.3-65
 [13] base64_2.0.2              scrime_1.3.5
 [15] magrittr_2.0.3            minfi_1.54.1
 [17] rmarkdown_2.29            yaml_2.3.10
 [19] ggtangle_0.0.7            doRNG_1.8.6.2
 [21] askpass_1.2.1             cowplot_1.2.0
 [23] DBI_1.2.3                 RColorBrewer_1.1-3
 [25] abind_1.4-8               quadprog_1.5-8
```

```
 [27] R.utils_2.13.0           purrr_1.1.0
 [29] RCurl_1.98-1.17          yulab.utils_0.2.0
 [31] GenomeInfoDbData_1.2.14  enrichplot_1.28.2
 [33] irlba_2.3.5.1            tidytree_0.4.6
 [35] rentrez_1.2.4            genefilter_1.90.0
 [37] pheatmap_1.0.13          annotate_1.86.1
 [39] dqrng_0.4.1              DelayedMatrixStats_1.30.0
 [41] codetools_0.2-20         DelayedArray_0.34.1
 [43] DOSE_4.2.0               xml2_1.3.8
 [45] tidyselect_1.2.1         aplot_0.2.8
 [47] UCSC.utils_1.4.0         farver_2.1.2
 [49] ScaledMatrix_1.16.0      beanplot_1.3.1
 [51] illuminaio_0.50.0        GenomicAlignments_1.44.0
 [53] jsonlite_2.0.0           multtest_2.64.0
 [55] survival_3.8-3           iterators_1.0.14
 [57] foreach_1.5.2            tools_4.5.1
 [59] treeio_1.32.0            Rcpp_1.1.0
 [61] glue_1.8.0               SparseArray_1.8.0
 [63] xfun_0.52                qvalue_2.40.0
 [65] dplyr_1.1.4              HDF5Array_1.36.0
 [67] withr_3.0.2              fastmap_1.2.0
 [69] rhdf5filters_1.20.0      openssl_2.3.3
 [71] digest_0.6.37            rsvd_1.0.5
 [73] gridGraphics_0.5-1       R6_2.6.1
 [75] colorspace_2.1-1         GO.db_3.21.0
 [77] RSQLite_2.4.2            R.methodsS3_1.8.2
 [79] h5mread_1.0.1            tidyr_1.3.1
 [81] data.table_1.17.8        rtracklayer_1.68.0
 [83] httr_1.4.7               S4Arrays_1.8.1
 [85] pkgconfig_2.0.3          gtable_0.3.6
 [87] rdist_0.0.5              blob_1.2.4
 [89] siggenes_1.82.0          XVector_0.48.0
 [91] htmltools_0.5.8.1        fgsea_1.34.2
 [93] scales_1.4.0             png_0.1-8
 [95] ggfun_0.2.0              knitr_1.50
 [97] rstudioapi_0.17.1        tzdb_0.5.0
 [99] reshape2_1.4.4           rjson_0.2.23
[101] nlme_3.1-168             curl_6.4.0
[103] bumphunter_1.50.0        cachem_1.1.0
[105] rhdf5_2.52.1             stringr_1.5.1
[107] KernSmooth_2.23-26       parallel_4.5.1
[109] restfulr_0.0.16          GEOquery_2.76.0
[111] pillar_1.11.0            grid_4.5.1
```

```
[113] reshape_0.8.10           vctrs_0.6.5
[115] BiocSingular_1.24.0      beachmat_2.24.0
[117] xtable_1.8-4             evaluate_1.0.4
[119] readr_2.1.5              GenomicFeatures_1.60.0
[121] cli_3.6.5                locfit_1.5-9.12
[123] compiler_4.5.1           Rsamtools_2.24.0
[125] rlang_1.1.6              crayon_1.5.3
[127] rngtools_1.5.2           labeling_0.4.3
[129] nor1mix_1.3-3            mclust_6.1.1
[131] fs_1.6.6                 plyr_1.8.9
[133] stringi_1.8.7            viridisLite_0.4.2
[135] BiocParallel_1.42.1      assertthat_0.2.1
[137] babelgene_22.9           Biostrings_2.76.0
[139] lazyeval_0.2.2           bspm_0.5.7
[141] GOSemSim_2.34.0          Matrix_1.7-3
[143] hms_1.1.3                sparseMatrixStats_1.20.0
[145] bit64_4.6.0-1            Rhdf5lib_1.30.0
[147] KEGGREST_1.48.1          statmod_1.5.0
[149] igraph_2.1.4             memoise_2.0.1
[151] ggtree_3.16.2            fastmatch_1.1-6
[153] bit_4.6.0                gson_0.1.0
[155] ape_5.8-1
```

Differential Methylation Analysis

Welcome! In this tutorial, we'll walk through the common steps of differential methylation analysis.

Download the data

First we will download the data from "GSE86297".

This study investigates the onset and progression of de novo methylation. Growing oocytes from pre-pubertal mouse ovaries (post-natal days 7-18) isolated and sorted into the following, non-overlapping size categories: 40-45, 50-55 and 60-65 m with two biological replicates in each.

```
#!/usr/bin/env bash
#
# Download the data from GEO
#
# ------------------------------------------------------------------------

wget https://ftp.ncbi.nlm.nih.gov/geo/series/GSE86nnn/GSE86297/suppl/GSE86297_RAW.tar

tar -xvf GSE86297_RAW.tar
```

The metadata of the samples is described as:

```
GEO Source Group File
GSM2299710 40-45um-A 40um GSM2299710_RRBS_40-45oocyte_LibA.cov.txt.gz
GSM2299711 40-45um-B 40um GSM2299711_RRBS_40-45oocyte_LibB.cov.txt.gz
GSM2299712 50-55um-A 50um GSM2299712_RRBS_50-55oocyte_LibA.cov.txt.gz
GSM2299713 50-55um-B 50um GSM2299713_RRBS_50-55oocyte_LibB.cov.txt.gz
GSM2299714 60-65um-A 60um GSM2299714_RRBS_60-65oocyte_LibA.cov.txt.gz
GSM2299715 60-65um-B 60um GSM2299715_RRBS_60-65oocyte_LibB.cov.txt.gz
```

Download the metadata from github

```
#!/usr/bin/env bash
#
# Download the metadata from github
#
# ------------------------------------------------------------------------

wget https://github.com/coriell-research/2025-coriell-summer-internship/raw/refs/heads/main/
```

The Bismark cov.txt.gz output format

The Bismark coverage files include the data for each sample representing methylation in the CpG context. These files will contain 6 columns describing the: chromosome, start position, end position, methylation proportion in percentage, number of methylated C's, and the number of unmethylated C's.

```
s1 <- read.delim(
  file="GSM2299710_RRBS_40-45oocyte_LibA.cov.txt.gz",
  header=FALSE,
  nrows=6
)

s1
```

```
> s1
V1 V2 V3 V4 V5 V6
1 6 3121266 3121266 0.00 0 17
2 6 3121296 3121296 0.00 0 17
3 6 3179319 3179319 1.28 1 77
4 6 3180316 3180316 4.55 1 21
5 6 3182928 3182928 4.33 22 486
6 6 3182937 3182937 5.37 61 1074
```

Reading in the data

To read in the data, we can use edgeR's helper function `readBismark2DGE`

```
md <- read.table("/path/to/metadata/MethylationMetadata.csv", sep = ",", header = TRUE)
rownames(md) <- md$Source
md$Filepath <- vapply(
  md$File,
  function(x) list.files(
    path = "/path/to/geo/data",
    pattern = x,
    recursive = TRUE,
    full.names = TRUE
  ),
  character(1)
)
yall <- edgeR::readBismark2DGE(md$Filepath, sample.names = md$Source)
```

The edgeR package will format this data such that in `yall$counts` for each sample you will have two columns called "Me" which corresponds to the methylated reads and "Un" which corresponds to the unmethylated reads. Each row correpsonds to a CpG locus found in the files. The genomic coordinates of the CpGs are stored in `yall$genes`. `yall$samples` corresponds to the metadata of the samples. It might be helpful to add the metadata you download to the `samples` component.

```
yall$samples <- cbind(yall$samples, md[md[rep(seq_len(nrow(md)), each = 2),],])
yall$samples$methylation <- rep(c("Me", "Un"), length(unique(yall$samples$Source)))
```

Filtering and Normalization

For this example we will remove the mitochondrial (MT) CpGs since they are typically not of interest.

```
table(yall$genes$Chr)
yall <- yall[yall$genes$Chr!="MT", ]
```

It might be useful to order your chromosomes in genomic order.

```
ChrNames <- c(1:19,"X","Y")
yall$genes$Chr <- factor(yall$genes$Chr, levels=ChrNames)
o <- order(yall$genes$Chr, yall$genes$Locus)
yall <- yall[o,]
```

You should also annotate your CpGs with the nearest gene transcription start site.

```
TSS <- nearestTSS(yall$genes$Chr, yall$genes$Locus, species="Mm")
yall$genes$EntrezID <- TSS$gene_id
yall$genes$Symbol <- TSS$symbol
yall$genes$Strand <- TSS$strand
yall$genes$Distance <- TSS$distance
yall$genes$Width <- TSS$width
```

The `Distance` column will describe the genomic distance in base pairs of the nearest gene to the CpG. If the distance is negative the TSS is upstream of the CpG and positive distances are downstream.

Before we can examine the methylation of the samples, we should remove any CpGs that have low coverage. In order to do this we should sum the Me and Un columns for each sample to obtain the total coverage for the loci.

```
Me <- yall$counts[,yall$samples$methylation == "Me"]
Un <- yall$counts[,yall$samples$methylation == "Un"]

Coverage <- Me + Un
```

To be conservative in our filtering we can make sure every sample has sufficient coverage for it to be included.

```
hasCoverage <- rowSums(Coverage >= 8) == 6
```

If you have an adequate number of CpGs to test, you may proceed. If the number of CpGs seems to low, maybe consider other filtering strategies.

We will also filer out CpGs that are never methylated or are always methylated because they will provide little information in the differential methylation and will dilute our testing.

```
hasBoth <- rowSums(Me) > 0 & rowSums(Un) > 0
```

You should check the table to get an understanding of how many CpGs fell into each category.

```
table(hasCoverage, hasBoth)
```

Now you can filter your DGEList for the desired CpGs to test in downstream analysis.

```
y <- yall[hasCoverage & hasBoth,,keep.lib.sizes=FALSE]
```

We need to calculate the actual library sizes since the reads are divides into the Un and Me categories.

```
TotalLibSize <- 0.5 * y$samples$lib.size[y$samples$methylation == "Me"] +
                0.5 * y$samples$lib.size[y$samples$methylation == "Un"]

y$samples$lib.size <- rep(TotalLibSize, each = 2)
```

Quality Control

To observe the methylation/coverage distributions for a given sample we can plot histograms of each sample.

You can look at the sample distribution for coverage to see if there might be any max coverage you wish to filter for. This might be indicated by a secondary peak at the high coverage ranges.

```
Me <- y$counts[, y$samples$methylation == "Me"]
Un <- y$counts[, y$samples$methylation == "Un"]
Coverage <- Me + Un

par(mfrow = c(2, 3))
for (i in md$Source) {
  hist(Me/Coverage, breaks = 100, main = i)
}

par(mfrow = c(2, 3))
for (i in md$Source) {
  hist(log10(Coverage), breaks = 100, main = i)
}
```

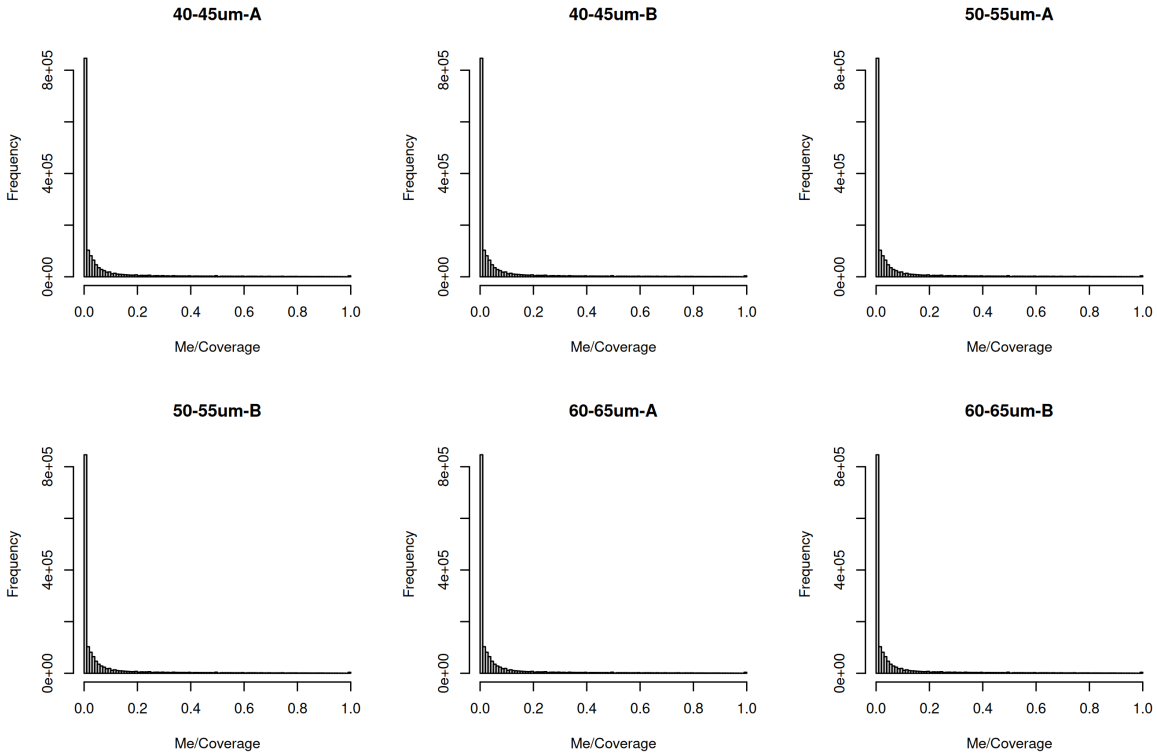


Figure 1: Histogram of Methylation

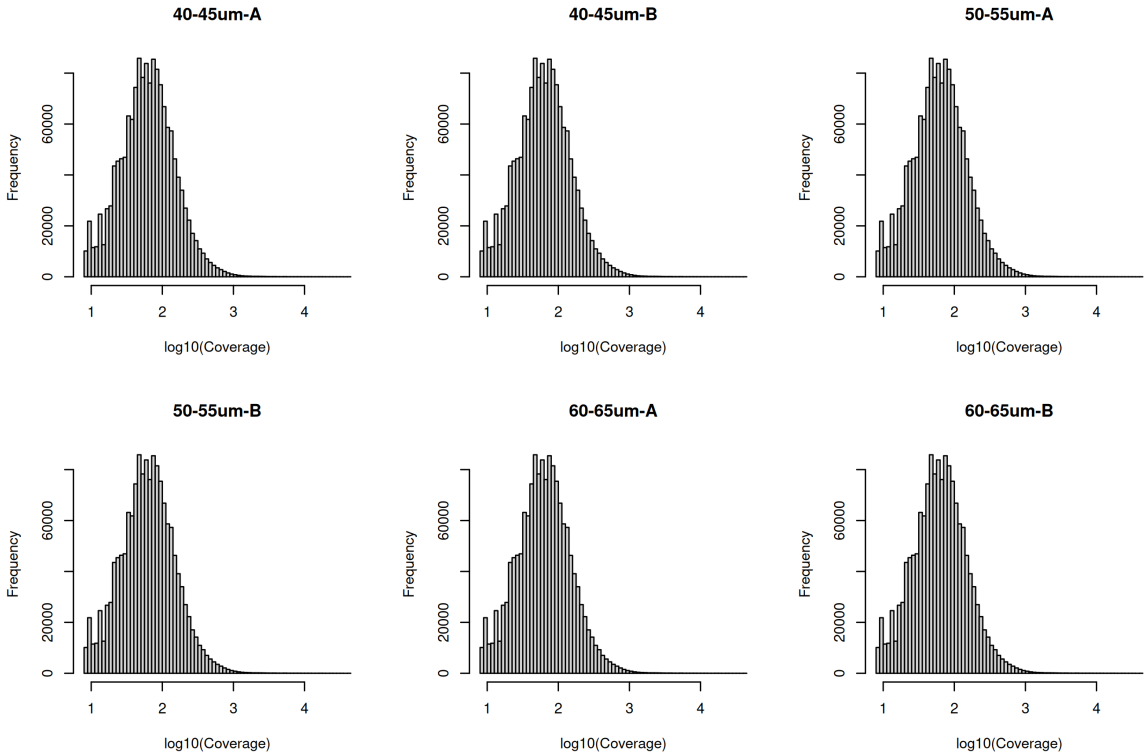
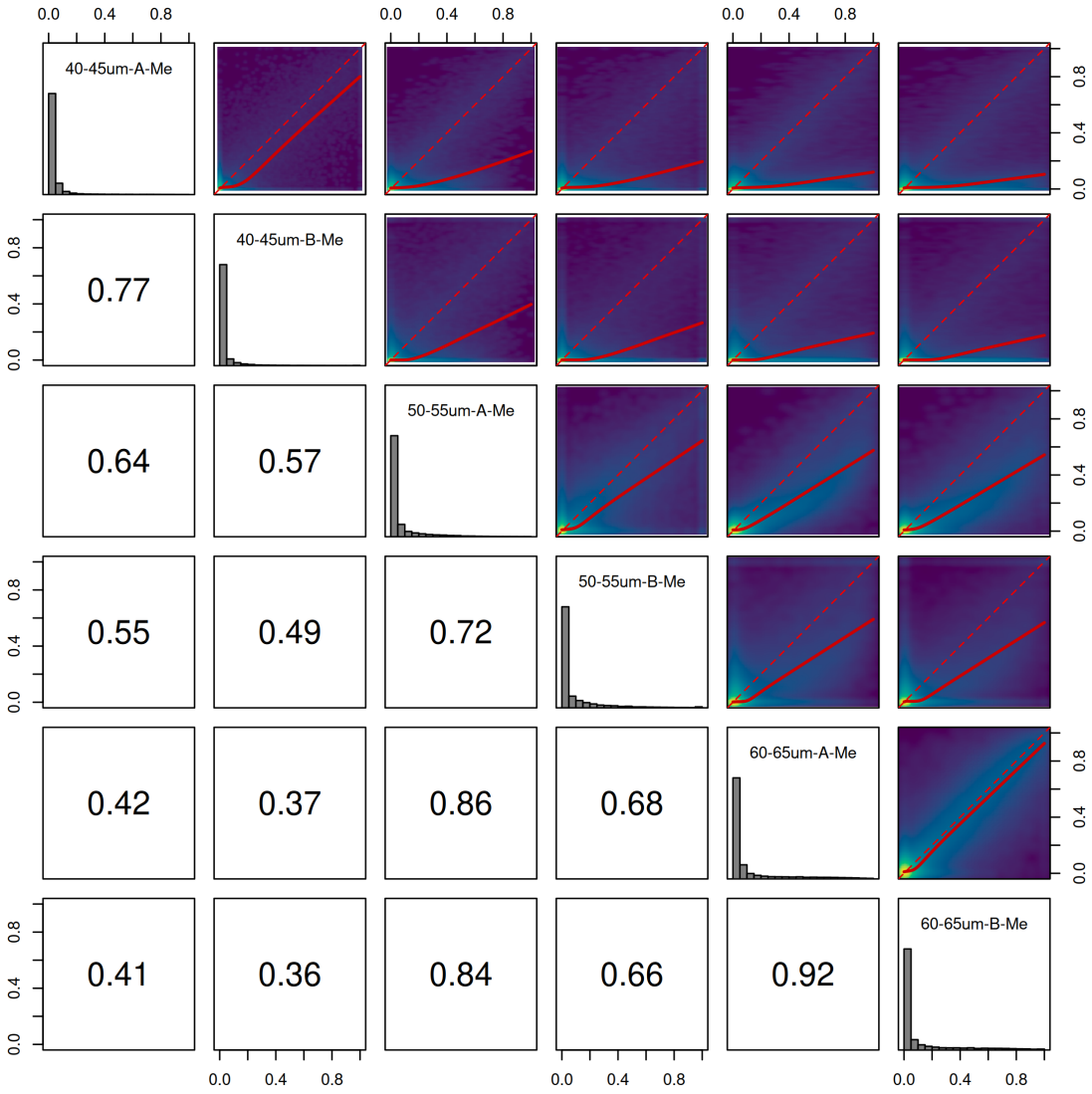


Figure 2: Histogram of Coverage

To observe sample concordance you can use the `plot_cor_pairs` function in the `coriell` package.

```
coriell::plot_cor_pairs(Me/Coverage, cex_labels = 1)
```



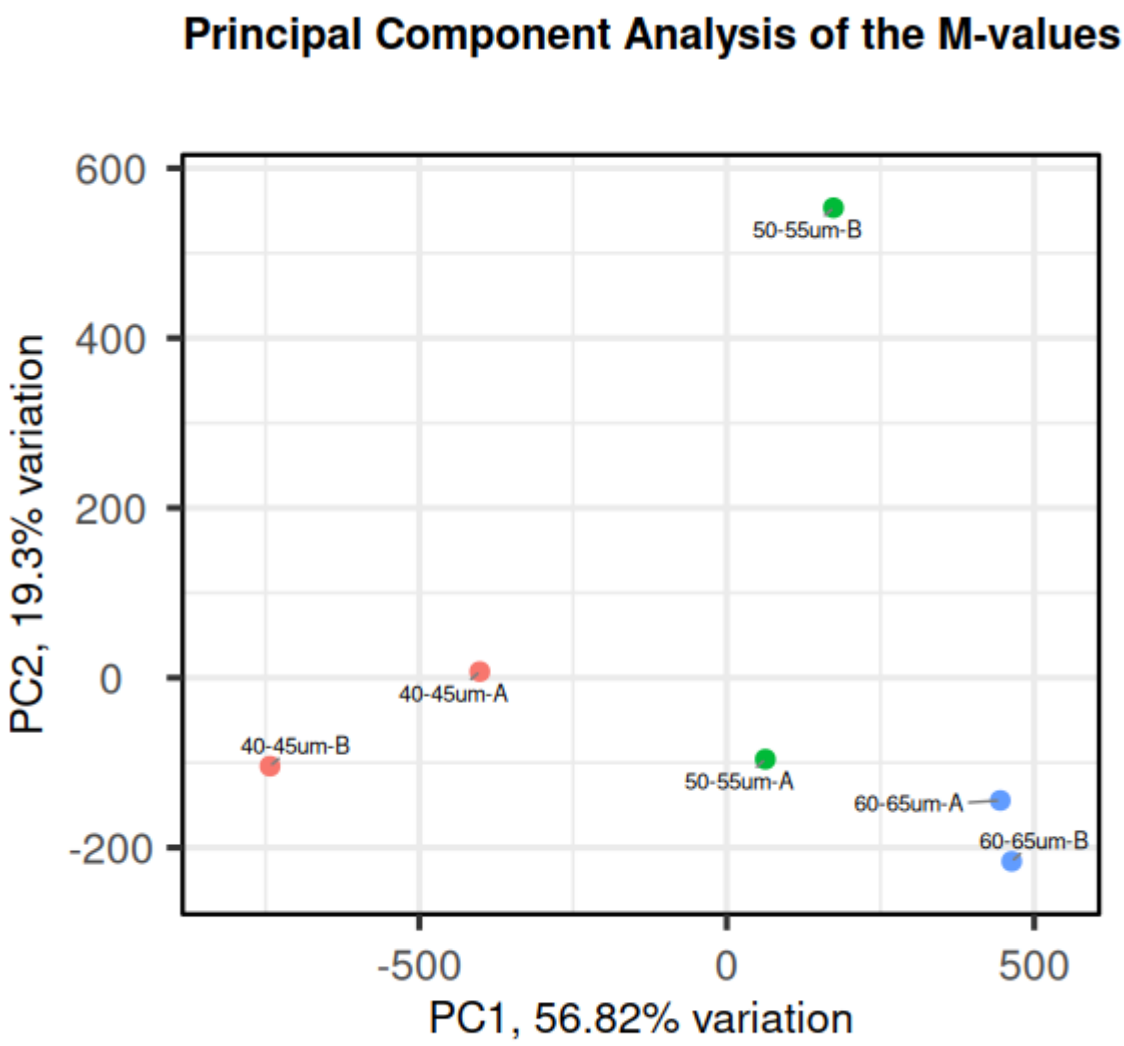
Data Exploration

The data can be explored by generating PCA plots of the methylation level in the M-value format. More information of the M-value can be found "here". An M-value is calculated by taking the log of the ratio of methylated and unmethylated C's. This is equivalent to the difference between methylated to the difference between methylated and unmethylated C's on the log-scale. A prior count of 2 is added to avoid taking the log of zero.

```
Me <- y$counts[, y$samples$methylation == "Me"]
Un <- y$counts[, y$samples$methylation == "Un"]
M <- log2(Me + 2) - log2(Un + 2)
colnames(M) <- md$Source
```

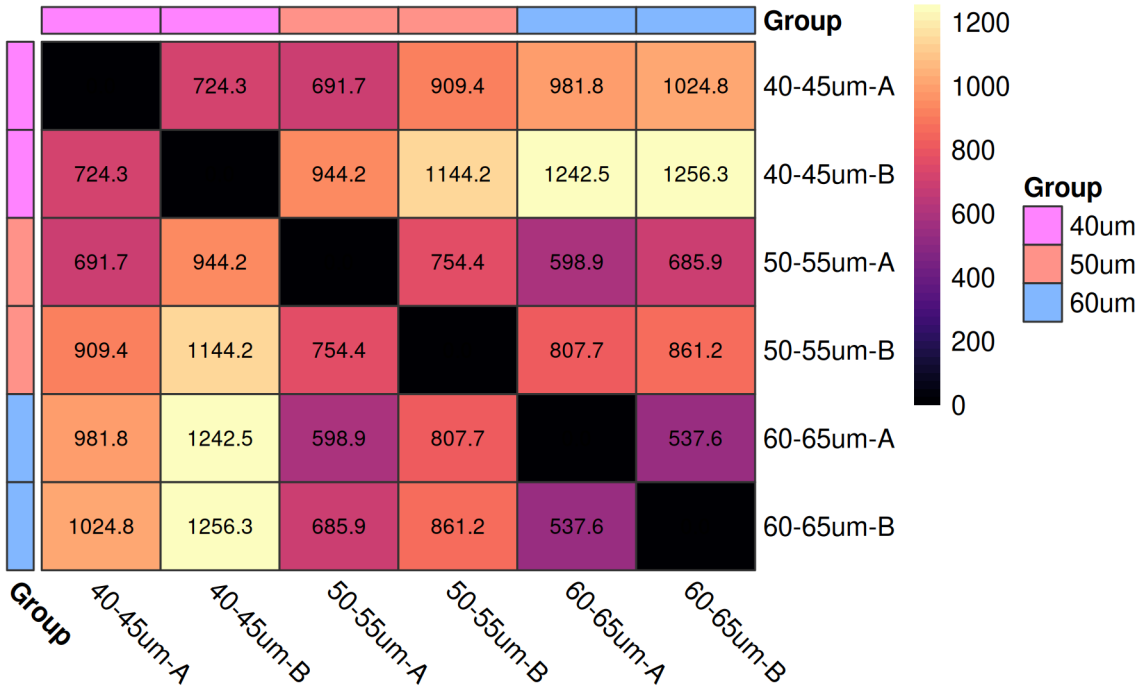
We can now use these M-values to generate a PCA biplot using the `PCAtools` package.

```
pca <- PCAtools::pca(
  M,
  metadata = md
)

PCAtools::biplot(
  pca,
  colby = "Group",
  title = "Principal Component Analysis of the M-values"
)
```



It might be of interest to observe the sample to sample distances as well.

```
coriell::plot_dist(M, metadata = md[, "Group", drop = FALSE])
```



Design Matrix

One aim of this study is to identify differentially methylated (DM) loci between the different cell populations. In edgeR, this can be done by fitting linear models under a specified design matrix and testing for corresponding coefficients or contrasts. A basic sample-level design matrix can be made as follows.

We then expand this to the full design which models sample and methylation effects.

The first six columns represent the sample coverage effects. The last three columns represent the methylation levels (in logit units) in the three groups.

```
designSL <- model.matrix(~0+Group, data=md)
design <- edgeR::modelMatrixMeth(designSL)
```

Differential methylation analysis at CpG loci

For simplicity, we only consider the CpG methylation in chromosome 1. We subset the coverage files so that they only contain methylation information of the first chromosome.

```
y1 <- y[y$genes$Chr==1,]
```

Then we proceed to testing for differentially methylated CpG sites between different groups. We fit quasi NB GLM for all the CpG loci using the glmQLFit function.

```
fit <- glmQLFit(y1, design)
```

We identify differentially methylated CpG loci between the 40-45 and 60-65 m group using the likelihood-ratio test. The contrast corresponding to this comparison is constructed using the makeContrasts function.

```
contr <- makeContrasts(Group60vs40 = Group60um - Group40um, levels=design)
qlf <- glmQLFTest(fit, contrast=contr)
```

The contrast object is a matrix showing the contrast desired

```
> contr
           Contrasts
Levels      Group60vs40
  Sample1             0
  Sample2             0
  Sample3             0
  Sample4             0
  Sample5             0
  Sample6             0
  Group40um          -1
  Group50um           0
  Group60um           1
```

We could also make multiple contrasts in one go and then select the contrast we wish to use like this.

```
contr <- makeContrasts(
  Group60vs40 = Group60um - Group40um,
  Group50vs40 = Group50um - Group40um,
  levels=design
)
qlf <- glmQLFTest(fit, contrast=contr[,"Group60vs40"])
```

```
> contr
           Contrasts
Levels      Group60vs40 Group50vs40
  Sample1             0           0
  Sample2             0           0
  Sample3             0           0
  Sample4             0           0
  Sample5             0           0
  Sample6             0           0
  Group40um          -1          -1
  Group50um           0           1
  Group60um           1           0
```

The top set of most significant DMRs can be examined with topTags. Here, positive log-fold changes represent CpG sites that have higher methylation level in the 60-65 m group compared to the 40-45 m group. Multiplicity correction is performed by applying the Benjamini-Hochberg method on the p-values, to control the false discovery rate (FDR).

```
topTags(qlf)
```

```
> topTags(qlf)
Coefficient:  -1*Group40um 1*Group60um
             Chr     Locus     logFC   logCPM        F       PValue          FDR
1-131987595    1 131987595 10.741404 2.698587 31.70873 1.850730e-08 0.0001409481
1-120170060    1 120170060  7.855076 4.531948 31.23248 2.362954e-08 0.0001409481
1-183357406    1 183357406  8.399456 4.272956 30.78670 2.970455e-08 0.0001409481
1-172206570    1 172206570 10.110386 2.706416 29.93463 4.601316e-08 0.0001637493
1-172206751    1 172206751 13.918907 1.294445 28.71671 8.607148e-08 0.0002302044
1-120169950    1 120169950  9.231213 3.469975 28.21772 1.112752e-07 0.0002302044
1-169954561    1 169954561 12.217222 2.352360 28.08869 1.189199e-07 0.0002302044
1-141992739    1 141992739 11.276043 1.335515 27.73221 1.428846e-07 0.0002302044
1-92943747     1  92943747  9.580566 2.358125 27.58701 1.539852e-07 0.0002302044
1-36500213     1  36500213  9.957566 2.195456 27.49192 1.617172e-07 0.0002302044
```

The total number of DMRs in each direction at a FDR of 5% can be examined with `decideTests`.

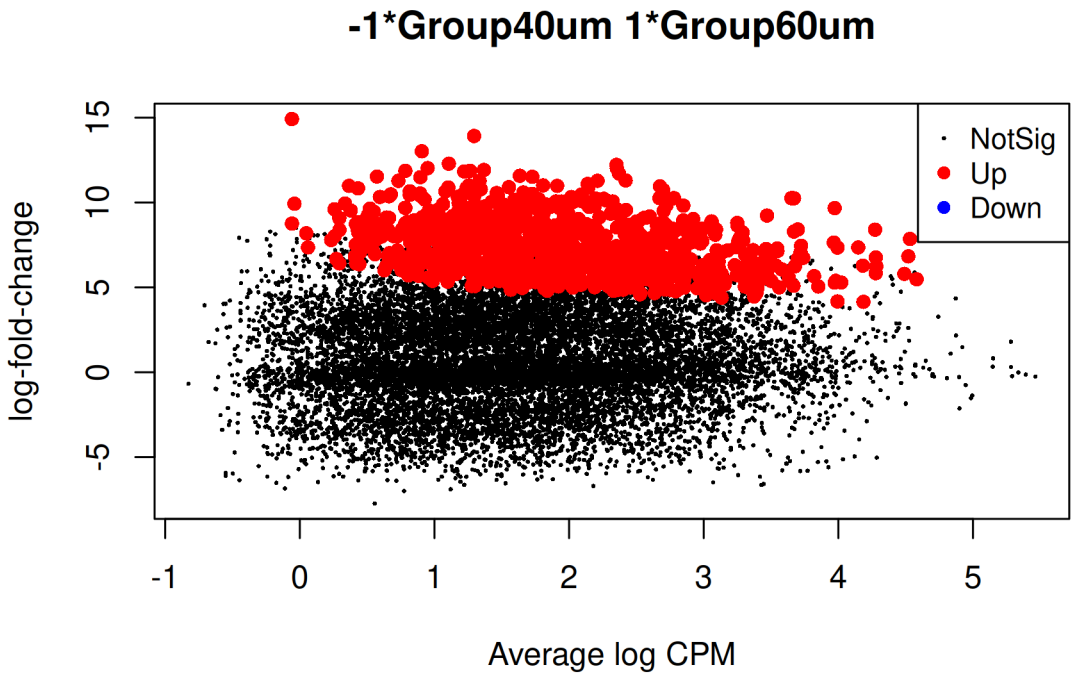
```
summary(decideTests(qlf))
```

```
> summary(decideTests(qlf))
       -1*Group40um 1*Group60um
```

```
Down                    0
NotSig              13252
Up                    983
```

The differential methylation results can be visualized using an MD plot. The difference of the M-value for each CpG site is plotted against the average abundance of that CpG site. Significantly DMRs at a FDR of 5% are highlighted.

```
plotMD(qlf)
```



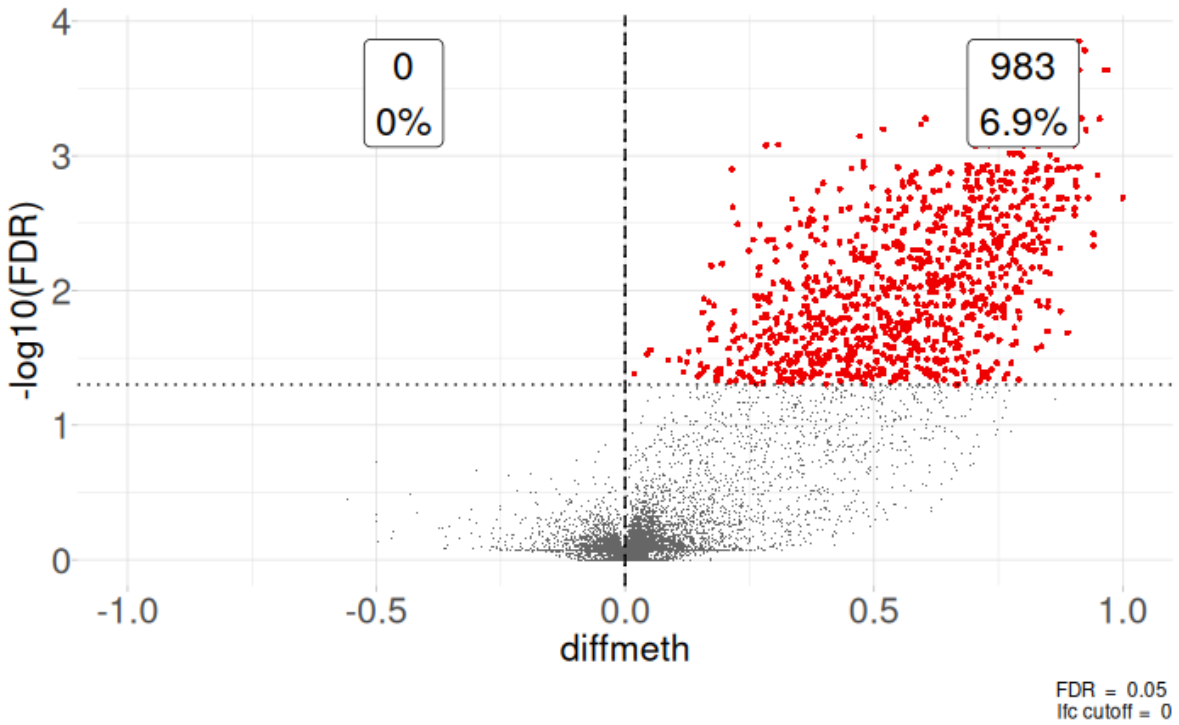
We can also plot volcanoes by calculating the difference in methylation of each group since log fold changes of these test might be hard to interpret as far as methylation levels go.

```
Me <- y1$counts[,y1$samples$methylation == "Me"]
Un <- y1$counts[,y1$samples$methylation == "Un"]
Coverage <- Me + Un
Beta <- Me/Coverage
colnames(Beta) <- rownames(md)

contr <- makeContrasts(
```

```
  Group60vs40 = Group60um - Group40um,
  levels=designSL
)
contr <- limma::contrastAsCoef(designSL, contr)
group1Methylation <- rowMeans(Beta[,which(contr$design[,"Group60vs40"] > 0)])
group2Methylation <- rowMeans(Beta[,which(contr$design[,"Group60vs40"] < 0)])
diffMethylation <- group1Methylation - group2Methylation

dataframe <- coriell::edger_to_df(qlf)
dataframe$group1 <- group1Methylation
dataframe$group2 <- group2Methylation
dataframe$diffmeth <- diffMethylation
```

Using these values we can plot a volcano plot using the `coriell` package.

```
coriell::plot_volcano(dataframe, x = "diffmeth", y = "FDR", lfc = 0, fdr = 0.05) +
  coord_cartesian(xlim = c(-1,1))
```



ATAC-seq Analysis

Work in progress

Filter raw fastq files

Use fastp to perform quality trimming on raw fastq files. Turn on adapter detection for paired end reads.

```
#!/usr/bin/env bash
#
# Run fastp on the raw fastq files
#
# ------------------------------------------------------------------------------
set -Eeou pipefail

FQ=/path/to/raw-fastq/00_fastq
SAMPLES=/path/to/sample-names.txt
OUT=/path/to/01_fastp
THREADS=12

mkdir -p $OUT

for SAMPLE in $(cat $SAMPLES)
do
    fastp -i $FQ/${SAMPLE}_R1.fastq.gz \
          -I $FQ/${SAMPLE}_R2.fastq.gz \
          -o $OUT/${SAMPLE}.trimmed.1.fq.gz \
          -O $OUT/${SAMPLE}.trimmed.2.fq.gz \
          -h $OUT/${SAMPLE}.fastp.html \
          -j $OUT/${SAMPLE}.fastp.json \
            --detect_adapter_for_pe \
          -w $THREADS
done
```

`sample-names.txt` is a plain text file listing the basenames for all samples. For example, if you have "sample1_R1.fq.gz", "sample2_R1.fq.gz", "sample3_R1.fq.gz" then `sample-names.txt` would be:

```
sample1
sample2
sample3
```

This file gets used throughout.

Perform alignment with Bowtie2

Use `--very-sensitive` mode and set max insertion size to 1,000. The output is piped to `samtools fixmate` for adding mate tag information and then to `samtools sort -n` to create name sorted BAM files.

It's assumed that you have a bowtie2 index generated for your species of interest. The number of jobs, threads, and memory allowed per thread will all be machine dependent.

```
#!/usr/bin/env bash
#
# Align trimmed reads with bowtie2, fixmate tags, and output name sorted BAMs
#
# ------------------------------------------------------------------------------
set -Eeou pipefail

FQ=/path/to/01_fastp
SAMPLES=/path/to/sample-names.txt
OUT=/path/to/02_align
IDX=/path/to/bt2_idx
INSERT=1000
JOBS=6
THREADS=4

mkdir -p $OUT

parallel --jobs $JOBS \
  "bowtie2 -x $IDX \
    -1 $FQ/{}.trimmed.1.fq.gz \
    -2 $FQ/{}.trimmed.2.fq.gz \
    --very-sensitive \
```

```
    --threads $THREADS \
    --maxins $INSERT | \
    samtools fixmate -m -@$THREADS - - | \
    samtools sort -n -@$THREADS -m4G -o $OUT/{}.bam - " :::: $SAMPLES
```

Call peaks with `Genrich`

Genrich is a fast peak caller designed to work with single samples or replicates. It has a dedicated option for peak calling ATAC-seq samples.

The options used below:

- Output pileups over called regions in a pseudo bedfile (-k)
- Output the log file (-f) which can be used to recall peaks with different -p and -a values
- Exclude regions in blacklist (-E)
- Exclude reads mapping to chrM (-e)
- Call in ATAC-seq mode (-j)
- Remove PCR duplicates (-r)
- Specify p-value threshold for peak calling (-p)
- Skip name sorting check (-S)

```
#!/usr/bin/env bash
#
# Call ATAC-seq peaks using Genrich on the name sorted BAMs
#
# -----------------------------------------------------------------------------
set -Eeou pipefail

SAMPLES=/path/to/sample-names.txt
BAM=/path/to/02_align
OUT=/path/to/03_callpeak
EXCLUDE=/path/to/excluderanges.bed
PVAL=0.05
JOBS=6

parallel --jobs $JOBS \
    "Genrich -t $BAM/{}.bam \
        -o $OUT/{}.narrowPeak \
        -k $OUT/{}.pileup.txt \
        -f $OUT/{}.log \
```

```
        -E $EXCLUDE \
        -p $PVAL -j -r -e chrM -S " :::: $SAMPLES
```

Exclusion lists

The "excluderanges.bed" file used above contains blacklisted regions of high-mappability that can be excluded from peak calling. This BED file can be created using the excluderanges R package. For example, to generate the excluded regions for mm39:

```
suppressMessages(library(GenomicRanges))
suppressMessages(library(AnnotationHub))

ah <- AnnotationHub()
query_data <- subset(ah, preparerclass == "excluderanges")
mm39_exclude_gr <- query_data[["AH107321"]]
rtracklayer::export.bed(mm39_exclude_gr, "mm39-excluderanges.bed")
```

Determine consensus peaks for replicate samples

After peak calling finishes, I create consensus peak calls for replicate samples. The consensus peak calls can be used in downstream differential accessibility analysis or for visualization. Creating consensus peak calls can be done in R by requiring all or some peaks to be called across replicate samples.

```
suppressPackageStartupMessages(library(here))
suppressPackageStartupMessages(library(GenomicRanges))


# Read in peak calls
peak_files <- list.files(here("data", "04_callpeak"), pattern="*.narrowPeak", full.names=TRU
names(peak_files) <- gsub(".narrowPeak", "", basename(peak_files))
peak_calls <- lapply(peak_files, rtracklayer::import)

# Separate into groups
control_calls <- peak_calls[c("control1", "control2", "control3")]
treatment_calls <- peak_calls[c("treatment1", "treatment2", "treatment3")]
control_calls <- GRangesList(control_calls)
treatment_calls <- GRangesList(treatment_calls)
```

```
# Compute coverage across ranges
control_coverage <- coverage(control_calls)
treatment_coverage <- coverage(treatment_calls)

# Determine regions with coverage in N replicates -- here require all 3 to have the same peal
control_covered <- GRanges(slice(control_coverage, lower=length(control_calls), rangesOnly=TR
treatment_covered <- GRanges(slice(treatment_coverage, lower=length(treatment_calls), ranges(

# Close gaps in the resulting regions -- min.gapwidth can be adjusted
control_consensus <- reduce(control_covered, min.gapwidth=101)
treatment_consensus <- reduce(treatment_covered, min.gapwidth=101)
all_consensus <- union(control_consensus, treatment_consensus)

# Export as BED files
rtracklayer::export.bed(control_consensus, here("data", "04_callpeak", "control-consensus.bed
rtracklayer::export.bed(treatment_consensus, here("data", "04_callpeak", "treatment-consensus
rtracklayer::export.bed(all_consensus, here("data", "04_callpeak", "consensus.bed"))
```

Create raw signal tracks

Genrich outputs a bedgraph-ish file using the `-k` flag that can be used to quickly create a a bigwig file of the raw signal which can be viewed in IGV. I find it useful to look at the raw signal tracks after peak calling so I can assess the absolute magnitude of the pileup that went into calling a peak. Then, if the peaks look too permissive or too stringent, adjust the peak calls.

Creating bigwigs from the bedgraph-ish files can be accomplished using `awk` and bedGraphToBigWig from UCSC.

```
#!/usr/bin/env bash
#
# Convert the pileups computed by Genrich from bedGraph-ish files to bigwigs
#
# ------------------------------------------------------------------------------
set -Eeou pipefail

SAMPLES=/path/to/sample-names.txt
PEAKS=/path/to/03_callpeak
OUT=/path/to/04_bg2bw
CHROM_SIZES=/path/to/chrom.sizes
JOBS=6
```

```
mkdir -p $OUT

echo "Extracting bedGraphs from experimental signal..."
parallel --jobs $JOBS \
  "awk 'NR > 2 { print \$1, \$2, \$3, \$4 }' $PEAKS/{}.pileup.txt | \
  LC_ALL=C sort -k1,1 -k2,2n > $OUT/{}.sorted.bedGraph" :::: $SAMPLES

echo "Creating bigwigs from pileups..."
parallel --jobs $JOBS "bedGraphToBigWig $OUT/{}.sorted.bedGraph $CHROM_SIZES $OUT/{}.bw" :::

echo "Cleaning up intermediate bedGraphs..."
find $OUT -type f -name "*.bedGraph" -delete
```

Chrom sizes files

You can supply a URL to `bedGraphToBigWig` to get the chrom.sizes if you're genome is supported. Otherwise, you can create a chrom.sizes file using another UCSC utility, `faSize`, and your genome fasta.

```
faSize -detailed -tab genome.fasta > genome.chrom.sizes
```

ATACSeqQC

TODO: complete this section

The ATACseqQC R package can be used to calculate various QC stats on the filtered BAM files. These should be assessed closely and compared to the ENCODE data standards.

Differential abundance using `csaw` and `edgeR`

TODO: elaborate on this section

csaw can be used to perform differential abundance over sliding windows *or* by counting reads in the consensus peaks defined above. This paper contains some code for performing and assessing ATAC-seq DA analysis using both methods.

I usually generate read counts over the consensus peaks using `featureCounts` from the Rsubread package. An SAF formatted file can be generated easily from the "consensus.bed" file. To get the read counts to align with the Genrich called peaks, set the `-read2Pos 5` flag to count alignments over the 5' end of reads.

Create TSS and region plots with `deeptools`

deeptools can be used to compute normalized coverage files in bigwig format for visualization in IGV. Normalization in `deeptools` can be really context dependent and what normalization strategy to use can often be unclear.

If you've performed differential abundance analysis and have global scaling normalization factors, they can be used at this step to generate TMM normalized bigwigs. In the absence of scaling factors, I've seen people use RPGC normalization or CPM for ATAC-seq.

Then you can run `deeptools` using the TSS regions of coding genes and called consensus peaks.

```
#!/usr/bin/env bash
#
# Compute normalized coverage bigwigs for visualization
#
# 'deeptools' mamba env must be activated before running
# ------------------------------------------------------------------------------
SAMPLES=/path/to/sample-names.txt
BLACKLIST=/path/to/excluderanges.bed
CODING=/path/to/coding.bed
BAM=/path/to/02_align
PEAKS=/path/to/consensus.bed
OUT=/path/to/05_deeptools
GENOME_SIZE=2654621783  # mm39 specific value, see https://deeptools.readthedocs.io/en/latest
JOBS=6
THREADS=4

mkdir -p $OUT

echo "Position sorting BAMs..."
parallel --jobs $JOBS "samtools sort -@$THREADS -m4G -o $OUT/{}.sorted.bam $BAM/{}.bam" ::::
parallel --jobs $JOBS "samtools index $OUT/{}.sorted.bam" :::: $SAMPLES

echo "Computing coverage over all BAM files..."
parallel --jobs $JOBS \
    "bamCoverage --bam $BAM/{}.sorted.bam \
                 --outFileName $OUT/{}.bw \
                 --outFileFormat 'bigwig' \
                 --normalizeUsing 'RPGC' \
                 --effectiveGenomeSize $GENOME_SIZE \
                 --binSize 1 \
```

```
                --extendReads \
                --blackListFileName $BLACKLIST \
                --numberOfProcessors $THREADS \
                --ignoreDuplicates" :::: $SAMPLES

echo "Computing matrix over TSS regions..."
computeMatrix reference-point \
 --regionsFileName  \
 --scoreFileName $OUT/*.bw \
 --outFileName $OUT/tss.mat.gz \
 --referencePoint TSS \
 --beforeRegionStartLength 3000 \
 --afterRegionStartLength 3000 \
 --blackListFileName $BLACKLIST \
 --missingDataAsZero \
 --numberOfProcessors $JOBS

echo "Computing coverage over peak BEDs..."
computeMatrix reference-point \
  --regionsFileName $PEAKS \
  --scoreFileName $OUT/*.bw \
  --outFileName $OUT/peak.mat.gz \
  --referencePoint 'center' \
  --beforeRegionStartLength 3000 \
  --afterRegionStartLength 3000 \
  --blackListFileName $BLACKLIST \
  --missingDataAsZero \
  --numberOfProcessors $JOBS

echo "Plotting heatmap of TSS enrichment..."
plotHeatmap -m $OUT/tss.mat.gz \
  -o $OUT/tss-heatmap.pdf \
  --dpi 300 \
  --colorMap "viridis" \
  --boxAroundHeatmaps 'no' \
  --samplesLabel "Ctl 1" "Ctl 2" "Ctl 3" "Trt 1" "Trt 2" "Trt 3" \
  --regionsLabel "Protein Coding Genes" \
  --yAxisLabel "RPGC" \
  --perGroup \
  --heatmapHeight 12 \
  --heatmapWidth 8 \
  --plotFileFormat "pdf"
```

```
echo "Plotting heatmap of MACS peak enrichment..."
plotHeatmap -m $OUT/peak.mat.gz \
  -o $OUT/peak-heatmap.pdf \
  --dpi 300 \
  --colorMap "viridis" \
  --boxAroundHeatmaps 'no' \
  --samplesLabel "Ctl 1" "Ctl 2" "Ctl 3" "Trt 1" "Trt 2" "Trt 3" \
  --regionsLabel "Consensus Peak Calls" \
  --refPointLabel "Center of Peak" \
  --yAxisLabel "RPGC" \
  --perGroup \
  --heatmapHeight 12 \
  --heatmapWidth 8 \
  --plotFileFormat "pdf"
```

Getting TSS coding region ranges

For ATAC-seq, I plot the enrichment over the TSS region for each sample as well as the coverage centered over the consensus peaks. To generate a BED file for protein coding gene regions ("coding.bed" above) in R:

```
# Get mouse annotation GTF, for example
url <- "https://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_mouse/release_M37/gencode.vM37.a
gtf <- rtracklayer::import(url)

# Extract the protein coding gene ranges only
coding <- gtf[gtf$gene_type == "protein_coding" & gtf$type == "gene", ]

# rtracklayer complains if score is NA or non-numeric
coding$score <- 1L

# Export to a BED file for use in deeptools
rtracklayer::export.bed(coding, here("doc", "gencode.vM37.coding.bed"))
```

Motif analysis with `MEME`

The MEME suite can be used to perform motif analysis on peaks that have been determined to have differential abundance. You can use R to generate fasta files for these regions based on GRanges extracted from DA analysis. These fasta files can then be used as input to `XSTREME`,

which performs de novo motif discovery, enrichment analysis, and comparisons with known motif databases.

For example, you can extract fasta regions like so (this uses hg38 as an example):

```
suppressPackageStartupMessages(library(data.table))
suppressPackageStartupMessages(library(Biostrings))
suppressPackageStartupMessages(library(BSgenome.Hsapiens.UCSC.hg38))
suppressPackageStartupMessages(library(GenomicRanges))


# Assuming you have a file from DA analysis that contains genomic coordinates
# of peaks and annotations - for example from ChIPseeker::annotatePeak()
peaks <- fread("annotated-peaks.tsv")
peaks <- makeGRangesFromDataFrame(peaks, keep.extra.columns = TRUE)

# Extract promoter peaks -- this can be any filtering you're interested in
promoter_peaks <- peaks[peaks$annotation %like% "Promoter"]

# Extract and write out the hg38 sequences as fasta records
pmtr_seqs <- getSeq(BSgenome.Hsapiens.UCSC.hg38, promoter_peaks)
writeXStringSet(pmtr_seqs, filepath = "promoter-peaks.fasta", format = "fasta")
```

These fasta records can then be used in `XSTREME` analysis from the MEME suite. The meme formatted databases of known motifs can be downloaded from MEME's website

```
#!/usr/bin/env bash
#
# Run XSTREME motif analysis on promoter peak sequences
#
# ------------------------------------------------------------------------------------------
FA=/path/to/promoter-peaks.fasta
DB=/path/to/HUMAN/HOCOMOCOv11_core_HUMAN_mono_meme_format.meme
OUT=/path/to/xstreme

xstreme --oc $OUT --p $FA --seed 123 --m $DB --no-pgc --dna
```

References