

HEINRICH HEINE UNIVERSITY, DÜSSELDORF

CLUSTER OF EXCELLENCE ON PLANT SCIENCES

RNAseq Workshop 2022

August 21, 2022

Alisandra Denton
Dominik Brillhaus

Contents

1	License	2
2	Acknowledgements	3
3	Preface	3
4	Course material	4
4.1	Annotated Research Context	4
4.2	During the workshop	4
4.3	Docker	5
4.4	After the workshop	6
5	Basics	7
5.1	Linux & Bash	7
5.2	R	12
5.3	Perl/Java/others	16
5.4	Python	17
6	Example short read RNAseq analysis	19
6.1	Description of the datasets you have been given to work on	19
6.2	First look at reads	19
6.3	Trimming	20
6.4	From Reads to Quantified Transcripts	20
7	Biological data extraction	26
7.1	Rstudio via second Docker file	26
7.2	The Plan	26
7.3	Import	26
7.4	Basic data.frame / matrix calculations	29
7.5	PCA and HCL	30
7.6	Differential Expression	32
7.7	GO term enrichment	35
7.8	MapMan	37
7.9	Adding information from public data sources	38
7.10	Further Clustering	41
8	Long Read Sequencing	45
8.1	The Plan	45
8.2	Resources	45
8.3	Data description	46
8.4	First Look	46
8.5	Trimming, Clustering and Polishing	47
8.6	Coding Genome Definition	49
8.7	Comparing and Evaluating Gene Models	59
8.8	Future perspectives	60
9	Questions and answers	61

1 License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License. To view a copy of this license, visit

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

In non-legal terminology: You are welcome to use, distribute, and modify this material for non-commercial purposes.

2 Acknowledgements

The authors would like to thank Juliane Schmid and the CEPLAS team for organizing.

This course builds directly or indirectly on a series of workshops that we've been involved with for over a decade. Particularly the 2018 iteration under the same name. Even though we do not have a list of contributors to the various workshops, we'd like to thank all of you all the same, and acknowledge this wasn't put together in a vacuum.

3 Preface

Here we want to jump start your big data and RNAseq analyses. We dive head first into doing this on the command line in linux (via Docker).

While there maybe more "user-friendly" options out there to complete the same tasks (google "galaxy rnaseq" if you're interested!), and there are certainly more precise and detailed deeper levels (e.g. algorithms and source code), we choose the command line for several reasons:

- it is customizable and flexible
- we work at a good level to gain a conceptual understanding
- you might as well try the command line in a setting where you can ask for help, you can always come back to galaxy & co. later.

At this point, we will not strive for the most elegant, nor the most efficient code; but rather for simplicity and understandability for beginners. We may write ugly, redundant, copy-paste code; but it will get the job done and allow us to analyze our own data following a more or less standardized pipeline.

There are many good resources out there, in so far as you find it useful we recommend using some of them, such as <https://linuxsurvival.com/>, to help cement what you have learned in the course. We also provide a set of handouts that may be useful to you during or after the workshop in the ARC under [_handouts/](#)

The Docker setup should also allow you to take home and practice the material anywhere you can get docker installed: <https://www.docker.com/>

4 Course material

This workshop is organized based on an ARC (Annotated Research Context).

4.1 Annotated Research Context

The ARC stores all required input data, backup data, scripts, presentations as well as this reader in one research data package.

4.1.1 The basic ARC structure summarized

- `studies` -> external data and sample metadata
- `assays` -> measurement data (i.e. "raw" sequence data) and metadata
- `workflows` -> computational analyses (i.e. "scripts")
- `runs` -> outputs of workflow (i.e. "results")

4.1.2 Material added to the ARC for this course

- `_reader` -> The basis to this reader (written in \LaTeX)
- `_slides` -> Slides presented during the workshop
- `_handouts` -> Cheat sheets and additional materials
- `runs/_backups` -> Backups for runs, in case a workflow unexpectedly did not work or takes too long.
 - If a workflow would output `blat_results`, but failed, copy `runs/_backup/blat_results` to `runs/blat_results`.

4.1.3 Disclaimer

For this workshop we do not take full advantage of all ARC features. This is in part due to the fact that some developments are work in progress. More so, we could easily spend two more days exploring those features, which are simply out of scope for a three-day workshop focusing on RNA-Seq data analysis. And besides, the ARC will in the future circumvent some annoyances that we run into for training purposes during this course.

If you want to learn more, check out the [DataPLANT website](#).

4.2 During the workshop

For the in-person workshop, we have already stored the ARC to the "Raumlaufwerke" folder under our room number (25.41.00.41). Please copy the folder "rnaseq-workshop" into your own home folder, so that everyone has their own copy. We have written all code and commands so that they can be "executed from" the root of the ARC.

As a check now, please open a terminal (applications -> terminal) and run:

```
cd $HOME/rnaseq-workshop
```

This is also a very good thing to try throughout the workshop if you're getting any sort of 'file not found' error, make sure you're in the right place.

4.3 Docker

We rely heavily on Docker for this workshop, which allows us to provide an 'image' with all the software you will need for the workshop installed and ready to use. There are two major reasons we chose Docker. 1) Convenience for development, as it works the same on our normal work machines, and on the ZIM teaching machines we use for the workshop, and more importantly 2) Portability, you can all take the Docker file home, work once through standardized instructions for installing Docker: <https://docs.docker.com/> (or ask your admin to), and then you can readily reproduce the work from the course.

As a brief disclaimer however, none of this is intended to demonstrate good practice with Docker, particularly not for other purposes.

The Docker image "rnaseq_docker.tar" is available on Sciebo at: <https://uni-duesseldorf.sciebo.de/s/53pA9W9Tb0KTgGQ>. the password is written on the board, or you have received it by e-mail.

The image can also be found in the "Raumlaufwerke" folder under our room (25.41.00.41).

4.3.1 load image

To load the image, run:

```
docker image load -i </path/to/>rnaseq_docker.tar}
```

Where `</path/to/>` is replaced by e.g. "Downloads" or the path to the "Raumlaufwerke" folder as appropriate.

4.3.2 run image as container

To start a writable 'container' from the image where you can work interactively, *from your home directory* (`cd $HOME`) run:

```
docker run -it --name rnalive -p 8889:8889 --mount \
  type=bind,source="$(pwd)"/rnaseq-workshop,target=/home/ZIM-gast/rnaseq-workshop \
  rnaseq:latest
```

4.3.3 restart container

```
docker start -i rnalive
```

4.3.4 exiting

You can exit a container with Ctrl+D or by typing `exit`.

You won't need to do this much in the workshop, however.

4.3.5 Then what?

You should now have a terminal, that looks a lot like the one before, except now know that you're in the 'container', where all the necessary software is installed.

```
# the workshop directory is shared
cd $HOME/rnaseq-workshop
# you should be able to see a list of all the files, that you can,
# e.g. by opening the folder in a file manager
ls
```

4.3.6 What to do when

At home:

- load: once per machine or update to the image
- run: once after each load, on first use
- start: whenever you want to use it

For the workshop however, the computers are wiped clean each night. So you will have to run :

- load: each morning
- run: after running load each morning
- start: as needed, should you exit the container

Important: keep any data you wish to save within the folder `rnaseq-workshop`, and make sure the contents of this folder are copied to either the "Raumlaufwerke" folder or to e.g. your USB flash drive or your sciebo account before you logout of the computer!!!

If you have any trouble whatsoever with Docker, please ask!!
Properly understanding docker is beyond the scope of this workshop, so don't worry if the above "doesn't make sense", but you will absolutely need it to be running as expected for the other parts to work, so just ask :-)

4.4 After the workshop

At least for the next half year, the ARC to this workshop is shared publicly under a CC-BY 4.0 license at <https://git.nfdi4plants.org/brilator/rnaseq-workshop.git>. Feel free to download and unarchive the whole ARC as a zip / tar archive or `git clone` or `git fork` the ARC. Just like docker, explaining the full usage of 'git' is beyond the scope of this course.

To ensure, that all code works as used in this reader, make sure to store the ARC as `rnaseq-workshop` in your \$HOME directory.

Similarly, the built docker image will remain available on sciebo. The built image is not included in the ARC as we did not have the time to check licenses for included software, regarding distribution.

5 Basics

5.1 Linux & Bash

5.1.1 Command-line basics

We type our instruction for the computer into the terminal, open one by typing (Ctrl + Alt + T) or by navigating to one using the GUI. In all languages used in the workshop, the # character indicates comments, and what follows will not be executed by the computer, but is there for the user.

```
# list the contents of the current directory
ls
# move to the home directory
cd
# list the contents
ls
# change to the directory with the data to be used
cd $HOME/rnaseq-workshop
# change to one directory up
cd ..
# make a directory and change into it
mkdir a_dummy_directory
cd a_dummy_directory
```

Alright, we can change between folders, look at their contents, etc... but we hardly needed to learn to use a command line to accomplish this. Let's very briefly look at some things that are nice on the command line, like variables, loops, and wild cards.

```
## variables
# set one variable
species="Arabidopsis_thaliana_Col0"

# use (here simply display) the variable by adding a '$' before the name
echo $species

# that might already save some typing, or make it easy to perform
# a similar analysis on multiple different species. But variables are
# even more useful when it comes to loops

## loops
# this will loop through the numbers 1 - 20
for i in {01..20};

# the keyword 'do' designates the start of each iteration
do
    # 'touch' will create an empty text file with the given name
    touch my_sample_${i}.txt

    # the keyword 'done' is used to designate the end of each iteration
done

# look at the result
ls

## wild cards
# we need one more file for the next example
touch keep_me_for_now.md
```

```
# note any differences in the results of the following commands

ls
ls *
ls *.md
ls my_sample_*

# now we can selectively clean up all those files we made
rm my_sample_*
ls # check result
# be careful with wild cards though, and when possible restrict
# their scope. Think about what would happen if you ran
# rm *
# in the wrong directory

## more on variables
# if you were wondering why we used ${i} and not $i in the loop
# the answer is it's best practice when the variable is used
# within words. Compare the results

echo $i
echo $isample.txt # '$isample' is not defined
echo ${i}sample.txt


# we'll also run into (and use) built-in variables
# for instance
echo $HOME # home directory
echo $PWD # working (current) directory

# we can return to our main directory this way
cd $HOME/rnaseq-workshop
```

Now we will write our first shell script. We will try everything out before we actually make the script. There are five steps to making a script: (i) define what you want to happen, in our case, we want the computer to print the current time and date, (ii) test out the code by writing it out piece by piece, (iii) actually write the script using a text editor, (iv) make the script executable and (v) run the script.

First, test the code:

```
# make the computer repeat what you wrote
echo "My 1st script prints the date and time."
echo $(date +%F_%T)
```

And now we will produce the script (or program if you will). Open the text editor, gedit, on the host machine either by double clicking a text file from the GUI file manager or e.g. by opening applications ( , lower left) and searching for 'gedit'.

Then write the following in the text editor:

```
#!/bin/bash
echo "My 1st script prints the date and time."
echo $(date +%F_%T)
exit
```

Save the file as `myfirstprogram.sh` in the `rnaseq-workshop` directory.

This program cannot yet be executed (ran) as the computer has not been told that it is an executable program. But we can set the permissions. We'll start just by checking the current permissions.


```
# from the rnaseq-workshop directory where you saved myfirstprogram.sh
ls
# note the colors
ls -l
# note the information you see on the screen
chmod u+x myfirstprogram.sh
# this command tells the computer to add (+) the
# execute permission (x) to the current user (u)
# if you omit the u, you will give permission to everyone
ls
ls -l
# compare what changed to output from above
```

As a side note, there's a lot more to permissions control, both in *what* can be done and also in *how* it can be done. It's more than we have time or need to get into here but there's a lot of nice resources out there to explain it if you're curious, e.g. <https://www.pluralsight.com/blog/it-ops/linux-file-permissions>.

You are ready to run the script.

```
# execute the script in the current directory (./)
./myfirstprogram.sh
# we can also store the output
./myfirstprogram.sh > text.txt
# look in the current directory in the GUI to find
# the file text.txt and open it by double clicking.

# further, view the output in the terminal
less text.txt
# enter 'q' to quit

# extra exercise:
# try appending output to the existing text.txt file
# by using '>>' instead of '>'
```

Writing shell scripts is easy, you just put in the program that you would write in the terminal and the computer will execute it line by line. The first line tells the computer what to use to read the program (in our case `bash`) and the last line tells it to exit from executing the program. Writing these small shell scripts becomes important when you want to run your read mapping overnight or over a weekend. We will revisit them when we do the read mapping.

You can also use commands to make analyses in a file. We will use a transcriptome (you will assemble your own later) which is in .fasta format

As necessary **C**hange into the **d**irectory

```
$HOME/rnaseq-workshop/studies/AthalianaReferences/resources/
```

```
# look at the file using the command line
less Athaliana_primaryTranscripts.fa
# you can move through the document by pressing Enter
# you can leave the file by pressing q
# if you only want to look at the first few lines
head Athaliana_primaryTranscripts.fa
# if you want to look at the last few lines
tail Athaliana_primaryTranscripts.fa
# view the whole file
cat Athaliana_primaryTranscripts.fa
# now you know why the 'head' and 'tail' commands
```

```
# are so important. To interrupt this, you'll
# want the shortcut Ctrl + C (or Strg + C)
```

With these large files, we often get our first look and first stats in bash.

```
# count the number of sequences, more specifically:
# count the headers, which start with '>'
# the grep command, is a type of search, we'll look for '>'
# the '|' tells the computer to pass the output of one command
# to another, and wc is the command for "word count"
grep ">" Athaliana_primaryTranscripts.fa | wc --lines

# the number you see, is the number of transcripts
# you can also estimate* how many bases there are in total
# by counting all characters in the file with the
# exception of the fasta headers.
# you do that by inverting grep (-v), producing all but
# lines starting with ">", and then counting characters
grep ">" Athaliana_primaryTranscripts.fa -v | wc --chars
# *estimate, because the end-of-line character is counted.
```

Linux has many more such small useful commands. It is frequently helpful to google or to look at Linux tutorials and see what you can scavenge for your purpose.

5.1.2 Installing software

If you're running behind, this is a good-to-know sub-section that we won't reference or need later in the course, so feel free to skip. You can always come back to it later on your own time.

We've pre-installed everything for this course, and we'll be sending you home with a copy of the Linux image. That said, we would be remiss if we didn't briefly cover how to install additional software.

In general, a lot of software can be installed through the package manager, which is best when the program is available there, up-to-date, and we have administrative privileges.

Let's try and install a program, "TopHat", which is a now-retired deprecated short-read aligner that we don't need, but which has a fairly typical installation process for a bioinformatics tool.

```
# search for a program
apt search tophat
# you can see (in green) exact program names matching the search
# now we know exactly what to install
sudo apt install tophat
# Hmmm... now we run into the problem that we don't have
# administrative rights on these computers
# That's OK though, you'll normally either
# a) have administrative rights yourself, or
# b) have a systems administrator around that you can ask for help
```

Note that the package-manager is distribution specific so the above 'apt' commands work for Ubuntu. While other systems have their own package-managers (e.g. Fedora uses 'dnf'). There are much more extensive resources on these elsewhere (e.g. <https://www.digitalocean.com/community/tutorials/package-management-basics-apt-yum-dnf-pkg>).

But now let's look at one fall-back alternative when installation with the package manager isn't working, basically how almost any compiled program can be locally "installed" for one user. This occurs a lot for bioinformatic programs, most often when packages aren't available via the package manager or they are only available in old versions (e.g. "FastQC", "samtools", "Trimmomatic"). That said, it can also work to

install software on e.g. a shared cluster, for instance when it's just a specific program that only you will use and you don't want to bother the administrators to install it.

Google "cufflinks rnaseq github" and your first hit is presumably <https://github.com/cole-trapnell-lab/cufflinks>.

Scroll down to "Installing a pre-compiled binary release", and click "here"

If you then right click the link for 2.2.1, Linux and then click "Copy Link" you get the path shown below. Or you could just left click it and download it normally of course, it would just slightly change the instructions from the demo.

You can further unpack the downloaded file by double clicking in the file manager GUI, or as shown below.

```
# change to a directory we can store the files
cd $HOME/rnaseq-workshop/workflows

# download and unpack the files
wget http://cole-trapnell-lab.github.io/cufflinks/assets/\
downloads/cufflinks-2.2.1.Linux_x86_64.tar.gz

tar -xvf cufflinks-2.2.1.Linux_x86_64.tar.gz
# look
ls
```

At this point we have downloaded and unpacked the file, but we can't just run it yet. The next thing one should almost always do is look for some sort of "Read me" file, (e.g. README.txt, Readme.txt, Readme.md) or installation instructions file (e.g. "INSTALL.txt").

Unfortunately Tophat's README doesn't do much but point you towards the website. But if you search there for 'install' or 'quick start' you should find some information. Even then, it's not exceptionally beginner friendly, so see below, if and when you want further help.

```
# change into the new cufflinks directory
cd cufflinks-2.2.1.Linux_x86_64
# look at what's there
ls -l
# you should see a lot of executable files
# check that tophat works
./cufflinks -h
```

Hopefully you got the help function and it's working. That's great, but this only works from this directory or if you supply the full path. e.g.

```
$HOME/rnaseq-workshop/workflows/cufflinks-2.2.1.Linux_x86_64/cufflinks .
```

That'd be a pain to remember. So how do we tell Linux to look for this executable no matter which directory we are currently in? We can do this through the `$PATH` variable.

This `$PATH` variable, stores all the directories where Linux will look for executables, and we can append directories to it.

```
# the syntax of the 'mv' AKA "move" command is 'mv source destination'
# now add the file to our path
export PATH=$PATH:$HOME/rnaseq-workshop/workflows/cufflinks-2.2.1.Linux_x86_64/
# to break this down
# PATH= is just setting the variable
# $PATH:$HOME/rnaseq-workshop/workflows/cufflinks-2.2.1.Linux_x86_64/ is just the
```

```
# old path with our custom bit appended (old):(custom)
# and 'export' makes it available to subprocesses of this shell

# test that it works
cufflinks -h
# test that something else still works
ls
# at this point, if you want this line to always be available, add
# the line "export PATH=$PATH:$HOME/Documents/tophat-2.1.1.Linux_x86_64"
# to the file $HOME/.bashrc and it will run every time you open a terminal
# e.g. using a text editor like 'gedit' as above
```

5.1.3 Getting help

Most programs and often even simple scripts will come with usage instructions.

```
# for instance, here are several ways to get documentation for 'grep'
man grep
grep -h
grep --help
# while the above commands are standard, you'll occasionally see things
# like -help or -?, and simply typing the name of the program without
# any parameters can also be a good bet

### challenge assignment ###
# grep can actually do a lot more than we showed here. From it's usage
# function, can you figure out how to accomplish the results of the
# command above: `grep ">" Athaliana_primaryTranscripts.fa | wc --lines`
# using just `grep`, and not `wc`?
```

If this doesn't help (enough) or perhaps you don't know quite which tool to use yet: Google it, duck-duck-go-it, your-search-engine-of-choice it.

Searching for any error messages you encounter can also be extremely helpful during trouble shooting.

5.2 R

We will run a brief introduction to R, including installing packages, getting your data into R, exporting data from R and how to make a figure using the package "ggplot2". We will use RStudio to write and execute R code since it makes our life easy. As a rule, Interactive Development Environments (IDEs), like RStudio, make the programmer's life easier, particularly at the start. PyCharm is a nice equivalent for python.

You can open Rstudio, either by entering `rstudio` on the command line, or from the applications menu under 'Development'.

5.2.1 Rstudio via second Docker file

Normally, you would open R by entering `rstudio` on the command line, or from the applications menu under 'Development'. However, for the duration of the course we'll use a second Docker container for this.

- acquire the `rstudio_docker.tar` file from sciebo or the "Raumlaufwereke" folder (as before)
- on the *host* machine, open a new terminal
- run the following

```
docker image load -i </path/to/>rstudio_docker.tar
docker run --rm -p 8787:8787 -e PASSWORD=rstudio --mount \
  type=bind,source=${PWD}/rnaseq-workshop,target=/home/ZIM-gast/rnaseq-workshop \
  rstudiotest
```

- on the *host* machine go to the browser and enter 'localhost:8787'
- enter 'rstudio' as the username and 'rstudio' as the password
- have fun learning R & Rstudio!

We will use the basic R functions that come with R when you install it. We will also use R packages, little or large things other people wrote for us and those need to be installed. To install one we type in the Rstudio script window and execute by typing (Ctrl + Enter). We type only the lines without the preceding "#" and we execute each line after typing it by pressing (Ctrl + Enter).

Within Rstudio, the first thing we'll want to do is navigate to the usual directory.

```
setwd('/home/ZIM-gast/rnaseq-workshop/')
```

5.2.2 basics in R and Rstudio

```
install.packages("ggplot2")
# this installs the packages, you can see some logging information in the
# console about what R is doing. If you get an error about permissions,
# don't worry, it's already installed
library(ggplot2)
# you only need to install the package once, afterwards you only need the
# library() command.
```

Let's assume you have an Excel file and want to do something to it using R. First, always, always use tab-delimited text files to move from program to program, no matter which programs you use. Look at the small tab-delimited txt file in the \$HOME/rnaseq-workshop/_WorkshopReader/intro_scripts_data called "example_table.txt" using a text editor (e.g. 'gedit') or LibreOffice Calc (which is just like Excel except free, open source, and Linux compatible). Next we'll import this into R

```
# set your working directory to the directory with the example_table.txt
# file using the [Session > set working directory > choose directory] GUI,
# and look what happens in the console
df0 <- read.delim("example_table.txt")
head(df0)
# the first command reads in a tab-delimited text file, our go-to format,
# it stores the table under the name 'df0'
# the second command, much like 'head' in bash shows the first rows
```

The standard format for biologists applications in R is the **data.frame** which is essentially what we used to call a table. It is organized in columns and rows with column names and row names (not present in our table, that is why `head(df0)` gives us numbers. Now we will briefly look at how to get particular columns out, and how to get particular rows out. We will always look at the data.frame, do something and look again. We will also look a bit at classes (or types) of data because that is a frequent cause for errors when writing scripts.

```
# let's remind ourselves what the data.frame looks like
head(df0)
# now we will extract the first column
df1 <- df0[, 1]
head(df1)
# now let's look at classes
class(df0)
```

```

class(df1)
# when we extracted the first column only, we no longer have a data.frame
# we now have a factor
# let's extract the first two columns
df2 <- df0[, c(1, 2)]
# this syntax means: take columns 1 and 2 from the data.frame 'df0'
# subsetting data frames works with [rows, columns]
# and c(1, 2) indicates which columns
head(df2)
df3 <- df0[, c("mpg", "cyl", "disp", "hp")]
head(df3)
# this time we have used the column names instead of the column number
# now let's look at rows
df4 <- df0[1:5, ]
# notice how the comma is now on the other side
# this extracts the first five rows, the column names are not a row, they
# are column names
df4
# notice how I no longer ask for head(), but show everything. After all,
# I now only have five rows to display
# now some advanced magic, extract all rows in which there
# are cars with at least six cylinders
df5 <- df0[(df0$cyl >= 6), ]
# this evaluates the stuff in the bracket which says check if column 'cyl'
# of the data.frame df0 is equal or larger than 6
# then it returns all rows (note the trailing comma) where this is TRUE
# how many?
dim(df5)
dim(df0)
df5

```

Let's assume you want to rename the columns but do not yet know how. If that happened in the lab, you'd ask a colleague or ask google. Same here: google "rename columns r". The best result is not always the first. To speed things up, we'll tell you the one we like, which is "How to rename a single column in a data.frame in R? - Stack Overflow" and there the 2nd solution which is:

```

# df = dataframe
# old.var.name = The name you don't like anymore
# new.var.name = The name you want to get
names(df)[names(df) == "old.var.name"] <- "new.var.name"

```

Now we can try renaming a column!

```

# look at current column names
colnames(df0)
# we tell R to look at the names in the data.frame df and find the one for
# which the name = mpg
# and then we re-assign that name to gas_mileage
# avoid spaces or number at the beginning of a name which are both bad in R
names(df0)[names(df0) == "mpg"] <- "gas_mileage"
# and look at after
colnames(df0)

```

In R, we can make plots and make prettier plots. There are three major options for plotting in R, ggplot2, lattice and the basic plot function. We will use the package ggplot2 which was installed and loaded earlier today. Our example plot will come from the data.frame we have worked with all the time. We

want to visualize whether horsepower of a car and gas mileage have something to do with each other. Let's look at the data.frame again:

```
head(df0)
# now we will learn about how ggplot makes figures
# the minimal information is where is the data, what is the x variable
# and what is the y variable and what we want to plot
# let's try

# this will make and save plot object
baseplot <- ggplot(data = df0, aes(x = hp, y = gas_mileage)) +
  geom_point()
# the first line indicates the data, what x is and what y is
# the second line says we want points

# this will display plot object
baseplot

# now we can customize that plot and make it prettier
# first off, different style, we do that by using theme
baseplot + theme_minimal() # display plot object + modification
# or alternatively, if you prefer
baseplot + theme_bw()
# now we'll save theme_bw with the object for later use
baseplot <- baseplot + theme_bw()

# now we want colored dots, simply red ones
# google ``r colors" for an idea of the named colors available in R
# you can choose them by name, number or hexcode; we like names
baseplot + geom_point(color = "firebrick")

# and a different size and shape
baseplot + geom_point(color = "firebrick", shape = 18, size = 4)

# or choose colors based on something else, like the number of cylinders

# note, that while many things can just be 'added' to a ggplot,
# a second ggplot object is not one of them, so we'll start over
# as it's normally best to specify dynamic color as an
# argument to aesthetics (called aes).
ggplot(data = df0, aes(x = hp, y = gas_mileage, color = cyl)) +
  geom_point(shape = 18, size = 4) +
  theme_bw()

# you can also make your color changes abrupt, not continuous,
# by defining cyl as a factor rather than a number
# first check what they are
class(df0$cyl)
# now make them a factor
baseplot <- ggplot(
  data = df0,
  aes(x = hp, y = gas_mileage, color = as.factor(cyl))
) +
  geom_point(shape = 18, size = 4) +
  theme_bw()
baseplot
# not the best colors, but you can modify them using scale_color_manual
```

```
# to do that you need to know which levels you have as your factors
levels(as.factor(df0$cyl))
# and now you can look at colors and decide what is what
baseplot <- baseplot +
  scale_color_manual(
    values = c(
      "4" = "black",
      "6" = "orange",
      "8" = "red3"
    ),
    name = "cylinders"
  )
baseplot
# finally, a little more polishing
baseplot +
  xlab("gross horsepower") +
  ylab("miles per gallon") +
  theme(text = element_text(size = 16))
```

You can spend days and weeks over figures in ggplot2. Refer to the cheat sheet provided by Rstudio for the most important functions.

<https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>

We will cover common RNAseq-related examples as needed.

5.3 Perl/Java/others

While they're common starting points and the most critical for this workshop, bash and R are just a tiny sampling of available languages. At the least, you'll want to know how to execute scripts in other languages. This is a quick primer on how to do that. Very generally, you start by checking whether the language is installed and you can do that either by typing the name and seeing what happens or by typing the name followed by `-v` or `--version` which usually checks the version or by typing the name followed by `-h` which will call the help pages.

```
perl
# nothing seems to happen, use the interrupt (Ctrl + C)
perl -v
# you should see the version information
perl -h
# shows included help function, -h, -help, and --help are your
# first "goto"s, the second is google
python3
# ups, you can now directly write code and execute it, we need
# to get out of here
quit
# this is not working but python tells you what will work
quit()
# and now you made it out.
# apparently both are installed
```

Google for "count_fasta.pl". This will give you a script. Look at the first line, apparently it is in perl (also standard for a .pl ending). Copy the script into the text editor and save it in the same folder as `Athaliana_primaryTranscripts.fa` naming it `count_fasta.pl`. Now it needs permission to be executed

```
ls
```



```
# the filename is white
chmod u+x count_fasta.pl
ls
# the filename is green
# find out how it works
./count_fasta.pl -h
# apparently, you can specify something using -i but you do
# not have to do it and you need a fasta file.
# Use the transcriptome file from earlier
./count_fasta.pl Athaliana_primaryTranscripts.fa
# this will compute for a moment and then give you statistics
# about the fasta file. If you wanted that as a text file,
# you can redirect the output into file using >
./count_fasta.pl Athaliana_primaryTranscripts.fa > ATstats.txt
# you can look at the resulting file using the GUI or by typing
less ATstats.txt
```

You can find scripts by googling what you want to do or what previous authors say they have used (or made) in papers. Hosting services such as <https://github.com> provide a way for all of us to backup, track and share our code. Many simple scripts stand alone and can be executed similar to above, while bundles of scripts or 'packages' may require installation. Just check whatever documentation comes with the code you want to use (it should contain instructions for any required installation) and give it a try. We will use Java later when we use trimmomatic and a lot of the analyses are performed with python scripts and packages.

5.4 Python

5.4.1 Python virtual environments

As more and more software is provided in python, it's more and more important to be at least partially familiar with installing things in python and with python virtual environments (yes including conda). Often, and particularly when installing less-stable bioinformatics software, there are good reasons to install it within a virtual environment. Basically it temporarily adds things to your PATH, while protecting you from any damage to your system that could be caused by, say, two executables having the same name.

We won't use a virtual environment for the primary software in this course, because we're using Docker.

However, a quick demo is worth the effort.

Virtual environments are also a great way to install things *with* a package manager *without* needing administrative rights. Pip is the general python package manager.

First, we'll setup a virtual environment

```
# setup environment, 'venv' can be any name or valid path of your choice
virtualenv venv
# activate or 'turn on'
source venv/bin/activate
```

Now we can use the virtual environment, and install things with a package manager.

```
# first, we'll try and run what we're about to install
jupyter
# no surprise there
pip install jupyter
# see all those individual progress bars whizzing past,
# those are all the dependencies that pip is taking care
# of for us
```

```
# start jupyter so that it's accessible from host
jupyter notebook --port 8889 --ip 0.0.0.0
# you can use 'Ctrl + c', and then 'y' to leave
# but leave it on until you've finished python basics
```

You can find much information on virtual environments at <https://docs.python-guide.org/dev/virtualenvs/>, or if you are doing a lot of Python work under windows or run into installation instructions that require `conda` you might be interested in Anaconda-based virtual environments <https://conda.io/docs/user-guide/index.html>.

Notes from a installation mini-workshop can also be found here https://github.com/weberlab-hhu/reproducibility-collection/blob/main/installation_intro/installation_intro.md

5.4.2 python basics

While it probably won't be critical for the rest of the workshop, we would recommend learning some python basics to everyone. It's a very approachable and widely used language.

If you have time, we recommend the following.

Above, when you ran 'jupyter notebook' it should have opened a server and listed a link to access it. Maybe something like `http://127.0.0.1:8889/?token=8f463d2...`, the one with 127.0.0.1 will be easiest.

Copy this link and paste it into a browser on your host machine.

Click on the 'new' button, and then make a new 'python3' notebook. Work through the examples here:

<https://jckantor.github.io/CBE30338/01.01-Getting-Started-with-Python-and-Jupyter-Notebooks.html>

Jupyter is also great for mixing code with documentation, visualization, or interpretation of results. A demo of what jupyter can do can be found here:

https://github.com/weberlab-hhu/reproducibility-collection/blob/main/jupyter_intro/jupyter_demo.ipynb

just as an example resources, there's much more.

When you're done with the above, hit Ctrl+c (Strg + c), followed by 'y' in the terminal to close the jupyter server.

6 Example short read RNAseq analysis

6.1 Description of the datasets you have been given to work on

The data you see is part of an experiment to test signaling during systemic acquired resistance (SAR) in *A. thaliana*. During this experiment you challenge a leaf with either mock or pathogen solution. Two days later, you harvest a different leaf for RNA-seq and see if the "we are under attack" signal has arrived and how it transforms the transcriptome. Your data is only from wild type *Arabidopsis* treated with mock and pathogen solution. The original experiment also contains the analysis of mutants with defects in different aspects of signaling, that through salicylic acid and that through pipelicolic acid. The complete experiment can be found here: <http://www.plantcell.org/content/28/1/102.short>

6.2 First look at reads

Many, but not all, bioinformatics tools support compressed data as input. If you do need to extract it, any GUI archive tool or quickly googling the ending of the file name (e.g. *.gz) with "extract Linux" should find an easy solution. For the workshop files, run

```
gunzip assays/Bernsdorff2016_Illumina/dataset/Col0treatment1.fastq.gz
```

Take a quick look at the fasta file

```
head assays/Bernsdorff2016_Illumina/dataset/Col0treatment1.fastq
```

Every four lines represents a read.

```
1: @ID
2: Sequence[ATCGN]
3: +ID
4: Phred quality scores
```

Note that if this was paired end data, each sample would have two files with both having matching sorting and read IDs with all the forward reads in one file and reverse reads in the other. The quality scores generally encode the numbers from 0-40 that are $-10 \log_{10} p$, where p =probability that the base call was incorrect. So 30 is a 1 in 1000, and 20 a 1 in 100 chance of a miss-called base. Currently, the most common encoding is Phred+33, and looks like this:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI
| | | | |
0..3.....26...31.....41
```

No one wants to look at fastq files by hand, instead, the program FastQC will create a full quality report for you.

```
mkdir runs/fastqc_results
fastqc assays/Bernsdorff2016_Illumina/dataset/Col0treatment1.fastq \
--outdir runs/fastqc_results
```

You can open the result in a browser. Either from a file manager, or from the terminal. *This has to be done from the host machine.*

```
firefox runs/fastqc_results/Col0treatment1_fastqc.html
```

FastQC uses a simple code of green for "good" or "normal" data and yellow then red for more questionable data. Go through the report items and compare them to good Illumina data:

https://www.bioinformatics.babraham.ac.uk/projects/fastqc/good_sequence_short_fastqc.html

and to bad Illumina data:

https://www.bioinformatics.babraham.ac.uk/projects/fastqc/bad_sequence_fastqc.html

How do they measure up?

6.3 Trimming

You've seen from the FastQC reports that the data both a) has lower quality towards the end of reads, and b) contains some adapter sequences. Both of these can lead to problems in down stream analyses, and the solution of choice is generally trimming the reads. Trimmomatic is a java program that can be used to perform both adapter and trimming steps. The command looks complicated, but for that it is a very powerful and flexible tool.

Breaking it down into pieces:

SE	indicates single end reads
Col0treatment1.fq	your file to trim
Col0treatment1.trimmed.fq	name of output file
ILLUMINACLIP:<adapters>:<seed mismatches>: <palindrome clip threshold>:<simple clip threshold>	this trims the adapters
MAXINFO:<targetLength>: <strictness (0-1 for longer-stricter)>	this trims low quality bases from the 3' end
MINLEN:<min>	drop reads below this length

```
# run trimmomatic
# note that the '\' in the following command is there to 'escape'
# the end-of-line character. If you write this on one line,
# skip the '\' character

mkdir runs/trimmed_fastq/

# $HOME/sw is just where trimmomatic is located in our Docker container
# and would need to match the machine in question
java -jar $HOME/sw/Trimmomatic-0.39/trimmomatic-0.39.jar SE \
  assays/Bernsdorff2016_Illumina/dataset/Col0treatment1.fastq \
  runs/trimmed_fastq/Col0treatment1.trimmed.fastq \
  ILLUMINACLIP:$HOME/sw/Trimmomatic-0.39/adapters/TruSeq3-SE.fa:2:30:10 \
  MAXINFO:50:0.8 MINLEN:36
```

It is normally best to check the quality of the data after trimming. This way you are confident the right adapters were removed.

```
fastqc runs/trimmed_fastq/Col0treatment1.trimmed.fastq --outdir runs/fastqc_results

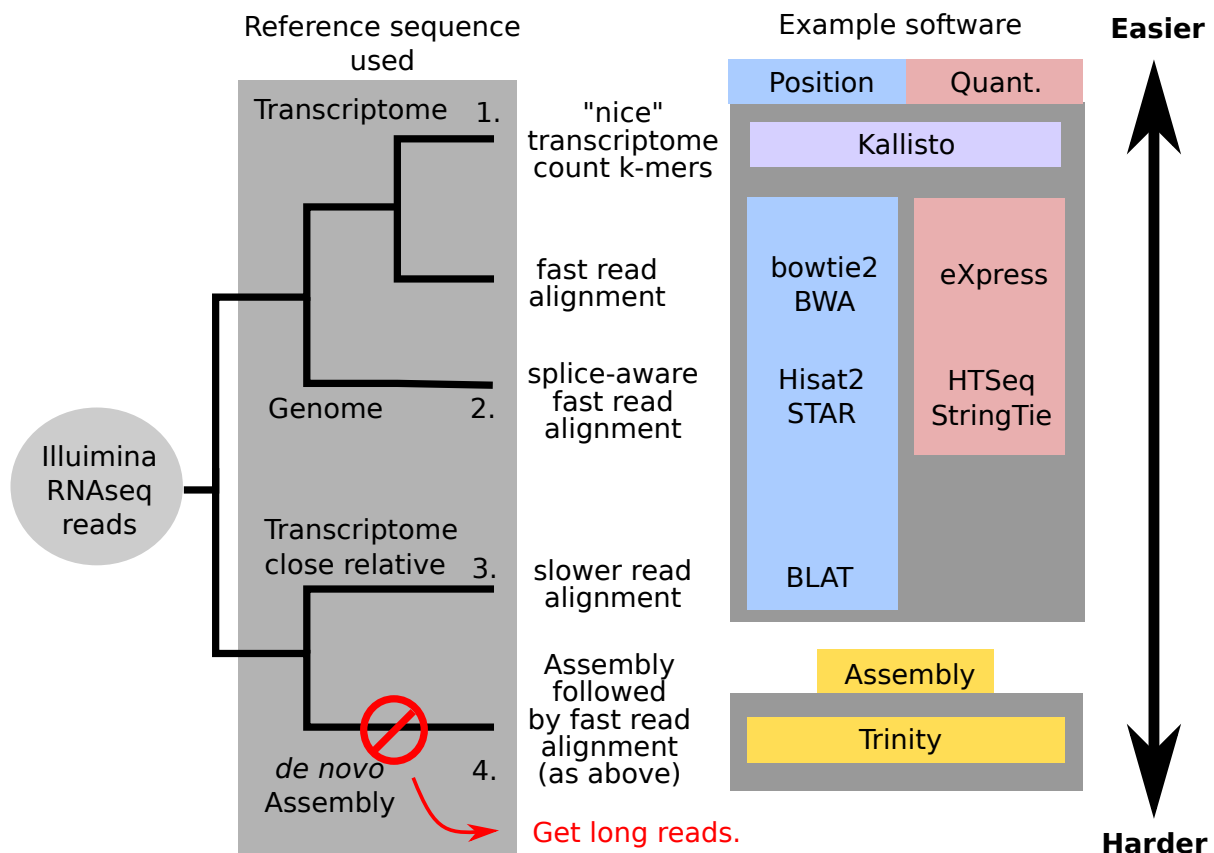
# and to view it
firefox runs/fastqc_results/Col0treatment1.trimmed_fastqc.html
```

6.4 From Reads to Quantified Transcripts

A file with 5 million reads, high quality or not, is still very far away from human legible data, let alone a biological conclusion. The next big step is to get from reads to transcripts and their abundance; however, what this looks like will depend heavily on what species you are using and how good of a genome and/or transcriptome annotation is already available.

Some major options are shown in the following figure. Basically, the less data is already available as a reference sequence, and the lower quality the available annotation there of, the harder this will become. If you have a nice genome and a well annotated transcriptome with only modest duplication, like *A. thaliana* used here. You can just align the reads to the transcriptome or even just count the k-mers in the reads. However, if you are working with a non-sequenced species, you will either have to align the reads to a closely related species, or make your own *de novo* assembly.

All of the options come with their own caveats, and what is appropriate depends on your reference and your data.



For instance:

- If your DNase treatment was incomplete, you don't want the DNA reads mapping to the most-similar transcript, you want to align the reads directly to the genome (including organellar genomes).
- If you have a species where the genome is more complete than the corresponding transcriptome annotation, your data will be easier to interpret if you align it directly to the genome.
- If your species of interest is not sequenced, and not phylogenetically close to a sequenced species, you will have little option but performing a *de novo* assembly, but should be aware of biases such as splitting highly abundant transcripts, chimeric transcripts, partial or absent low-abundance transcripts, and detection of transcripts from other species (e.g. biotrophic fungi). Getting long reads will help with some, but not all of the above.
- If everything has gone well and you have a good genome available, a k-mer counting strategy will run very quickly and very easily and give you data you can work with.

We will focus on option 1. from the figure using Kallisto for k-mer counting, as we are working with clean, easy, *A. thaliana* data. However, we'll also run a brief example for branches 2. - 4., in case this more accurately reflects your data.

6.4.1 Counting k-mers with kallisto

Classically reads have been aligned to the reference sequence. While it can be sped up by efficient indexing, this is fundamentally a computationally intensive (and therefore slow) process, because the reference sequences are very large and there are many reads to search against them. Recently, options such as Kallisto have been developed, that do not actually align the reads, but count the k-mers they are made up of and assign these counts to transcripts with these k-mers. A k-mer is a stretch of bases from a

read long enough to generally be unique, short enough to rarely contain errors and to be computationally feasible.

Example sequence and 6-mers that can be derived from it:

TCTTCCGGAGGTGGAGGAAAACCGACGATT

TCTTCC

CTTCCG

TTCCGG

TCCGGA

CCGGAG

.

.

.

While much faster than read alignment, this will sometimes be less accurate for assigning reads to the correct transcripts as it does not fully utilize all the read information.

When available, plant reference sequences can be obtained from major databases like Phytozome (phytozome.jgi.doe.gov) or Ensembl (plants.ensembl.org), which provides some naming consistency. Many more species are available through species specific resources.

Here we have pre-downloaded the *A. thaliana* information from Phytozome.

```
# first kallisto needs to create an index (de Bruijn graph)
# of the reference sequence
mkdir runs/kallisto_index/

kallisto index -i runs/kallisto_index/Ath_kallisto_index \
studies/AthalianaReferences/resources/Athaliana_transcripts.fa
```

The indexing parameters are:

index	tells kallisto to make an index
-i	assign the index name
Athaliana_transcripts.fa	the downloaded reference transcriptome

```
# we'll want to keep our output in a separate folder here
mkdir runs/kallisto_results

# now kallisto can quantify how many reads (appear to) come
# from each transcript

kallisto quant -b 30 -i runs/kallisto_index/Ath_kallisto_index \
-o runs/kallisto_results/treatment1 --single \
-l 190 -s 20 runs/trimmed_fastq/Col0treatment1.trimmed.fastq
```

The general quantification parameters are:

quant	tells kallisto to run quantification
-b	number of bootstraps (optional, for use with sleuth)
-i	index name assigned above
-o	output directory name

And a few extra parameters are required for single end reads:

<code>--single</code>	
<code>-l</code>	mean fragment length
<code>-s</code>	standard deviation of fragment length

```
# kallisto created a directory with three files
ls runs/kallisto_results/treatment1

# we just need one of them, take a look
less runs/kallisto_results/treatment1/abundance.tsv
```

You should see tab separated values, most importantly showing us the transcript ID (target_id), the estimated read counts (est_counts), and the normalized abundance estimate Transcripts Per Million (tpm).

target_id	length	eff_length	est_counts	tpm
ATCG00020.1	1062	873	995	293.647
ATCG00040.1	1581	1392	9	1.66579
ATCG00050.1	240	51.331	0	0
ATCG00065.1	114	5.2474	0	0

We will later use the TPM for graphs and the counts for statistics. However, you may have noticed that we've so far ignored five of our six read files. You can save all the commands individually to a file for each of the other samples and run it.

```
#!/bin/bash
# trimmomatic

java -jar $HOME/sw/Trimmomatic-0.39/trimmomatic-0.39.jar SE \
assays/Bernsdorff2016_Illumina/dataset/Col0treatment2.fastq.gz \
runs/trimmed_fastq/Col0treatment2.trimmed.fastq \
ILLUMINACLIP:$HOME/sw/Trimmomatic-0.39/adapters/TruSeq3-SE.fa:2:30:10 \
MAXINFO:50:0.8 MINLEN:36

java -jar $HOME/sw/Trimmomatic-0.39/trimmomatic-0.39.jar SE \
assays/Bernsdorff2016_Illumina/dataset/Col0treatment2.fastq.gz \
runs/trimmed_fastq/Col0treatment3.trimmed.fastq \
ILLUMINACLIP:$HOME/sw/Trimmomatic-0.39/adapters/TruSeq3-SE.fa:2:30:10 \
MAXINFO:50:0.8 MINLEN:36

# ... and so on
# kallisto
kallisto quant -i runs/kallisto_index/Ath_kallisto_index \
-o runs/kallisto_results/treatment2 --single \
-b 30 -l 190 -s 20 runs/trimmed_fastq/Col0treatment2.trimmed.fastq

kallisto quant -i runs/kallisto_index/Ath_kallisto_index \
-o runs/kallisto_results/treatment3 --single \
-b 30 -l 190 -s 20 runs/trimmed_fastq/Col0treatment3.trimmed.fastq
# ... and so on.
```

However, if you recall loops from the bash introduction, we can use loops, which will almost certainly be easier in the long run. Particularly if you have to change a parameter later. Here is a very simple loop to catch all five of the other samples up with treatment1

```
#!/bin/bash
# for each word following "in" the loop will be ran with
# this word saved in the variable $sample
```

```

for sample in treatment2 treatment3 mock1 mock2 mock3
do # open the loop

    echo "running sample: $sample"
    java -jar $HOME/sw/Trimmomatic-0.39/trimmomatic-0.39.jar SE \
        assays/Bernsdorff2016_Illumina/dataset/Col0${sample}.fastq.gz \
        runs/trimmed_fastq/Col0${sample}.trimmed.fastq \
        ILLUMINACLIP:$HOME/sw/Trimmomatic-0.39/adapters/TruSeq3-SE.fa:2:30:10 \
        MAXINFO:50:0.8 MINLEN:36

    kallisto quant -i runs/kallisto_index/Ath_kallisto_index \
        -o runs/kallisto_results/${sample} --single \
        -b 30 -l 190 -s 20 runs/trimmed_fastq/Col0${sample}.trimmed.fastq

done # close the loop

```

Ideally, you will save the loop to a file, this is extremely helpful when it comes to repeating your work, taking on the next study, or writing the methods section.

Now we have six directories with six abundance files. We'll put these together into organized tables later, but first let's take a look at some of the other options for when k-mer counts is not applicable.

6.4.2 Aligning reads to the genome

This is a very robust method, even in combination with a good reference it can clean up the results compared to using a primary transcriptome. For instance, it allows inclusion of splice variants without causing problems with ambiguous alignments for reads that map to shared exons. Also, in reality most RNAseq will have a lot of reads that properly map to intergenic regions, be they from DNA contamination, unannotated genes, rRNA, or less-than perfect transcriptional regulation. That and reads from introns that have not yet been spliced. Including the genomic background in the reference allows reads to find their best mapping and doesn't bias these towards the next-closest gene. This all comes at a cost of run time, but in the worst case it comes down to the computers time vs your time trying to interpret the data later.

Recent developments in indexing have also greatly sped up the alignment stages. In particular Hisat2 makes a 10-fold gain on speed, while producing highly similar results to its predecessor, TopHat.

In contrast to k-mer counting with Kallisto, alignment and quantification is a two step process, so we'll first align reads with Hisat2, and then count the reads uniquely mapping to each locus with HTSeq.

```

# make the index
mkdir runs/hisat_index/

hisat2-build studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa \
    runs/hisat_index/Ath_hisat2_index

# align the reads
mkdir runs/hisat_results/

hisat2 -x runs/hisat_index/Ath_hisat2_index \
    -U runs/trimmed_fastq/Col0treatment1.trimmed.fastq \
    -S runs/hisat_results/treatment1.hisat.sam

# quantify the alignments
htseq-count runs/hisat_results/treatment1.hisat.sam \
    studies/AthalianaReferences/resources/Athaliana_167_TAIR10.gene_exons.gtf \
    -s no > runs/hisat_results/treatment1.hisat.htseq

```



```
# check output
less runs/hisat_results/treatment1.hisat.htseq
```

For information on the various parameters, check the help function of the individual programs.

If you wanted to use this method, you could run a script with a loop in a very similar way to what we did with Kallisto.

6.4.3 Aligning reads to the transcriptome of a related species

Sometimes your species of interest hasn't been sequenced, but one that is phylogenetically close has. Because *de novo* transcriptome assemblies can be very hard to work with, cross species alignment can be a very appealing option. Bear in mind, you won't get the same resolution between paralogs that you would have mapping to the actual genome, and the alignment will take much longer.

We will demo BLAT (BLAST-like Alignment Tool), which falls in between BLAST and a short-read aligner like Bowtie2 on the sensitivity vs speed scale.

```
# Still, on these computers we'll be using a smaller read set
# (minidata.fastq), and we'll have to convert it to fasta
mkdir runs/miniexample/

awk 'NR % 4 == 1 {print ">" $0 } NR % 4 == 2 {print $0}' \
studies/BLATexample/resources/minidata.fastq > runs/miniexample/minidata.fasta
```

You might be wondering what `awk` is and why that command looked so much more *complicated* and less *human readable* than most? Basically, `awk` makes it easy to perform simple, custom, text manipulations. It's very useful when what you want to do is a bit too complicated for some combination of what we've seen before such as `find` (e.g. `grep`) or `find` and `replace` (e.g. `sed`), but is not yet complicated enough that one wants to write a python script.

As a brief explanation, the command above checks the remainder when the number of rows (NR) is divided by 4 in order to determine if it's on an ID or sequence row in a fastq file. It appends '>' to the ID and just prints the whole sequence, producing fasta format.

A proper introduction to `awk` is more than we have time for here, but there's a lot of nice resources available on-line, e.g. <https://www.tutorialspoint.com/awk/>. In any case, you can use the commands here, or simply search for and use `awk` code for what you want to accomplish, without already having mastered `awk`.

```
## now we set up a BLAT database (not required, but faster)
# we will map reads to Brassica rapa transcripts
mkdir runs/blat_db/

faToTwoBit studies/BrapaReferences/resources/Brapa_primaryTranscripts.fa \
runs/blat_db/Brapa.2bit
## run blat (with 6-frame translation)
mkdir runs/blat_results/

blat runs/blat_db/Brapa.2bit runs/miniexample/minidata.fasta -out=blast8 \
-q=dnax -t=dnax runs/blat_results/treatment1-Brapa.tsv
## count up the results
# count_blat.py is a mini-script to count the
# best blat/blast hits to each target sequence.
workflows/count_blat.py -i runs/blat_results/treatment1-Brapa.tsv > \
runs/blat_results/treatment1-Brapa.counts
# check output
less runs/blat_results/treatment1-Brapa.counts
```

7 Biological data extraction

7.1 Rstudio via second Docker file

Please see sub section "Rstudio via second Docker file" from "Basics" on the first day to restart working environment.

7.2 The Plan

Now, we use R in earnest. Rstudio will let us see what is happening immediately when we do it, so it is a good environment for beginners.

Note: To allow for the level of customization that is often necessary for each project, this section is a little lower-level and relies less on pre-made tools than before. In as much there is a lot of R code, and retyping it would be rough on the schedule, so we'll be using a different paradigm than we did with the bash scripts. You don't have to retype anything, rather you're encouraged to copy from the .pdf. It is then your responsibility to go through it slowly and make sure you understand what you are doing. We would recommend, for every section to make sure you have understood it, that you vary something(s). Look at a different column, change the plot color, etc... and then save your changes into a script.

If for some reason we haven't given you the .pdf yet, please notify us.

We begin by getting our data from the kallisto results into a data.frame in R. Open a GUI which allows you to see the data structure – all results are in folders with different names, but the files themselves all have the same name.

7.3 Import

```
# we'll start with the mock1 results, as an example
first <- read.delim("runs/kallisto_results/mock1/abundance.tsv")
# and look at the first lines of the resulting data.frame
head(first)
# and check the dimensions, we expect one row per gene, and 5 columns
dim(first)
# we need new names, so that we know est_counts and tpm are from mock1
colnames(first)[4:5] <- c("mock1_estcounts", "mock1_tpm")
head(first)

# OK, for the second file, mock2, we'll do the same
second <- read.delim("runs/kallisto_results/mock2/abundance.tsv")
# we again need new names, so that we know est_counts and tpm are from mock2
colnames(second)[4:5] <- c("mock2_estcounts", "mock2_tpm")
# and look at the first lines of the resulting data.frame
head(second)
# some of the columns (length, eff_length) we only need once
# so we'll keep just the IDs (for merging) and the abundance
second <- second[, c(1, 4, 5)]
# and look again!
head(second)

# now we'll merge the tables and store the result in a data.frame called 'dfr'
dfr <- merge(first, second, by = "target_id", all.x = T)
# merge combines the first two arguments by the column with the name
# specified with the 'by'. Setting all.x=TRUE makes
# sure that the original data.frame stays complete
head(dfr)
```

```

# now onto "mock3", but we don't want to retype this every time
# so we'll change a few things, 1st we'll save the sample_name
# so we can type it once and use it more
sample_name <- "mock3"
# for instance, we can use the sample_name to enter the file name
paste0("runs/kallisto_results/", sample_name, "/abundance.tsv")
# and now actually import the file
newfile <- read.delim(paste0(
  "runs/kallisto_results/",
  sample_name, "/abundance.tsv"
))
head(newfile)
# 2nd, the code above is "hidden-bug prone", imagine (or test) what happens if
# we'd run these two lines from above out of order, or we'd run them twice
# ---
# colnames(second)[4:5] <- c("mock2_estcounts", "mock2_tpm")
# second <- second[,c(1,4,5)]
# ---
# so we'll use a more robust version from now on
# we'll subset the table by name instead of number
newfile <- newfile[, c("target_id", "est_counts", "tpm")]
# we'll check the old names when renaming the columns
colnames(newfile)[names(newfile) == "est_counts"] <- paste0(sample_name, "_estcounts")
colnames(newfile)[names(newfile) == "tpm"] <- paste0(sample_name, "_tpm")
# merging remains the same
dfr <- merge(dfr, newfile, by = "target_id", all.x = T)
head(dfr)

# for the remaining three samples all we have to do is change the sample name
sample_name <- "treatment1"
newfile <- read.delim(paste0("runs/kallisto_results/", sample_name, "/abundance.tsv"))
newfile <- newfile[, c("target_id", "est_counts", "tpm")]
colnames(newfile)[names(newfile) == "est_counts"] <- paste0(sample_name, "_estcounts")
colnames(newfile)[names(newfile) == "tpm"] <- paste0(sample_name, "_tpm")
dfr <- merge(dfr, newfile, by = "target_id", all.x = T)

sample_name <- "treatment2"
newfile <- read.delim(paste0("runs/kallisto_results/", sample_name, "/abundance.tsv"))
newfile <- newfile[, c("target_id", "est_counts", "tpm")]
colnames(newfile)[names(newfile) == "est_counts"] <- paste0(sample_name, "_estcounts")
colnames(newfile)[names(newfile) == "tpm"] <- paste0(sample_name, "_tpm")
dfr <- merge(dfr, newfile, by = "target_id", all.x = T)

sample_name <- "treatment3"
newfile <- read.delim(paste0("runs/kallisto_results/", sample_name, "/abundance.tsv"))
newfile <- newfile[, c("target_id", "est_counts", "tpm")]
colnames(newfile)[names(newfile) == "est_counts"] <- paste0(sample_name, "_estcounts")
colnames(newfile)[names(newfile) == "tpm"] <- paste0(sample_name, "_tpm")
dfr <- merge(dfr, newfile, by = "target_id", all.x = T)
# and we look at the result
head(dfr)

# now some ordering of the result
# we are going to use the function 'grep' for this, which is
# a search function that returns the index(es) where a pattern was found.
# syntax: grep(pattern_to_look_for, item_to_search_in)

```

```

grep("target", colnames(dfr))
grep("tpm", colnames(dfr))
grep("unicorn", colnames(dfr))
# we want the columns with general information
new_order <- c(
  grep("id|length", colnames(dfr)),
  # followed by the columns with "estcounts"
  grep("estcounts", colnames(dfr)),
  # followed by the columns with "tpm"
  grep("tpm", colnames(dfr))
)
# double check that was the order we wanted
new_order
colnames(dfr)[new_order]
# and now change the table
dfr <- dfr[, new_order]
head(dfr)
dim(dfr)
# if it's easier for you, remember you could have done this more manually
# dfr <- dfr[, c(1, 2, 3, 4, 6, 8, 10, 12, 14, 5, 7, 9, 11, 13, 15)]

# now export the table in biologist readable format
dir.create(path = "runs/kallisto_combined/", recursive = T, showWarnings = F)
write.table(dfr,
  file = "runs/kallisto_combined/mothertableV1.txt",
  row.names = F, sep = "\t", quote = F
)

### importing data 2 ###
# that was fun, and you want to do it again, right?
# OK, technically you can skip this part, your data is imported.
# If, however, you want to import dozens or hundreds of samples,
# you will want to use loops. This example should return the
# exact same result as above.

# find and save names of files to import
files <- dir("runs/kallisto_results")
# the first file gets special handling,
# since we want to keep the "length" related columns
sample_name <- files[1]
dfr <- read.delim(paste0("runs/kallisto_results/", sample_name, "/abundance.tsv"))
colnames(dfr)[names(dfr) == "est_counts"] <- paste0(sample_name, "_estcounts")
colnames(dfr)[names(dfr) == "tpm"] <- paste0(sample_name, "_tpm")
# then we run the code in the block for all but the first file

for (sample_name in files[-1]) {
  newfile <- read.delim(paste0(
    "runs/kallisto_results/",
    sample_name, "/abundance.tsv"
  ))
  newfile <- newfile[, c("target_id", "est_counts", "tpm")]
  colnames(newfile)[names(newfile) == "est_counts"] <- paste0(
    sample_name,
    "_estcounts"
  )
  colnames(newfile)[names(newfile) == "tpm"] <- paste0(sample_name, "_tpm")
}

```

```

dfr <- merge(dfr, newfile, by = "target_id", all.x = T)
} # the right curly bracket closes our loop

new_order <- c(
  grep("id|length", colnames(dfr)),
  grep("estcounts", colnames(dfr)),
  grep("tpm", colnames(dfr))
)

dfr <- dfr[, new_order]

# the loop makes for less code that is easier to modify and maintain
# but it's more cryptic at first, for your projects, it's your call!
# we will be emphasizing basic and readable for the workshop

# one last setup item, we will occasionally need not the transcript but the gene IDs
# the gene (locus) ID is simply the first nine characters of an AGI
locus <- substr(dfr$target_id, 1, 9)
dfr <- cbind(locus, dfr)

```

The results from both methods are identical. This first step by step method is more to write, but safe for beginners since you can change the names by hand. The loop is more challenging to produce but certainly faster to write. We will use the laborious way for the rest of the training course.

7.4 Basic data.frame / matrix calculations

Before we do high level analyses, let's calculate means for the replicates and fold-changes.

```

dfr <- read.table(
  file = "runs/kallisto_combined/mothertableV1.txt",
  sep = "\t", header = T
)

# let's add convenience columns with averages and fold changes
# R has a built in function 'rowMeans'
# all we need to do is pass what we want to take the means of
mock_tpm <- dfr[, grep("mock._tpm", colnames(dfr))] # '.' is a wildcard
dfr$mean_mock <- rowMeans(mock_tpm)
head(dfr) # check your work
# and again for 'treatment'
treatment_tpm <- dfr[, grep("treatment._tpm", colnames(dfr))]
dfr$mean_treatment <- rowMeans(treatment_tpm)
head(dfr)

# OK, what if we want to calculate something that isn't built into R?
# like log2 fold change?
# we can write our own function
log2FC <- function(denominator, numerator) {
  log2((numerator + 1) / (denominator + 1))
} # +1 so we don't divide by 0!

# always sanity check your functions / code!
log2FC(40, 80) # test where the ~ answer is obvious
log2FC(dfr[1, "mean_mock"], dfr[1, "mean_treatment"]) # test on our data
# then we apply our function to our data
# in a way 'apply' is a distant cousin of 'fill' in Excel, syntax:
# apply(data, 1, some_function) # fill down

```

```

# apply(data, 2, some_function) # fill across
# apply(data, c(1, 2), some_function) # fill down and across
dfr$log2FC <- apply(
  dfr[, c("mean_mock", "mean_treatment")], 1,
  function(x) log2FC(x[1], x[2])
)
# and look at the result
head(dfr)

# we can store the data in biologist readable format by using write.table
write.table(dfr,
  file = "runs/kallisto_combined/mothertableV2.txt",
  row.names = F, sep = "\t", quote = F
)
# and we can store the data as an R object
save(dfr, file = "runs/kallisto_combined/mothertableV2.Rdata")

# we've accumulated some variables, it's clean up time
ls() # or look under the 'Environment' tab
# remove variables we won't need any more
remove(
  new_order, newfile, sample_name, first, second, files, mock_tpm,
  treatment_tpm, locus
)
ls()

```

7.5 PCA and HCL

Before we start, let's think about what to expect. We have an experiment with a single factor, the treatment. So when we look at how the samples group together, how many groups with information do we expect? Yes, right, only one. So for our principal component analysis we expect that we have one dimension reflecting the treatment variation. As we have no other variables, the other dimensions of the PCA should show "noise" or biological variation due to random variables outside of our control. Ideally, we want to see the treatment variation in the first dimension (meaning most of the variation is due to treatment) and the noise based variation should occupy the lower dimensions. Now we start with our first analysis. We ask whether the replicates are more similar to each other than the samples. To that we use principal component analysis and hierarchical clustering of the samples.

```

## PCA
# we need a few plotting-related libraries we have not used before
library(ggrepel)

load(file = "runs/kallisto_combined/mothertableV2.Rdata")

# we need the normalized data for this analysis, the columns with tpm
for_clust <- dfr[, grep("tpm", colnames(dfr))]
head(for_clust)
# we'll filter to leave only, more abundant, always-above-0 genes
for_clust <- for_clust[apply(for_clust, 1, max) > 100, ]
for_clust <- for_clust[apply(for_clust, 1, min) > 0, ]
# and log2 transform
for_clust <- log2(for_clust)
# the prcomp (PCA) function assumes samples are in rows, and genes columns
# so we will transpose our data
for_clust <- t(for_clust)
# and we will scale it (z-score: mean center, and divide by standard deviation)

```

```

for_clust <- scale(for_clust)

# now we can do the PCA
pca <- prcomp(for_clust)
# and look at the outcome
s <- summary(pca)
s
# in row two, we see the 1st PC explains about 55% of the variance
# let's make a plot! As with most things in R, there's many ways to plot,
# we'll be using the package "ggplot2", it has some overhead, but it's
# faster to make a publication-quality plot in.
# first we extract the values we need for the plot from the pca object
scores <- as.data.frame(pca$x)
# now we add a column indicating what is mock and what is treatment.
scores$treatment <- rep(c("mock", "treatment"), each = 3)
# and we look if you data.frame is as expected
scores

# now we produce the figure with ggplot
# look at the beginning of the course or on the cheatsheet to figure out
# what the different lines mean
# the line labs() is difficult to understand, it is a fancy way to directly
# get the percentages in the plot without looking them up and writing them in

pca12 <- ggplot(data = scores, aes(x = PC1, y = PC2)) +
  theme_bw() +
  geom_point(aes(color = treatment), size = 2.5) +
  geom_text_repel(aes(label = treatment), size = 3) +
  labs(
    x = paste0("PCA1 (", s$importance[2, 1] * 100, "%)"),
    y = paste0("PCA2 (", s$importance[2, 2] * 100, "%)"),
  ) +
  ggtitle("PCA") +
  theme(legend.position = "none")
# with the next line we make Rstudio show us the figure
pca12

# <<< challenge assignments >>> #
# 1. modify the above plot to compare PC1 with PC3
# 2. comment out elements, to see what each does
# 3. use the cheatsheet, google and your intuition to replace
# the individual labels with a joint legend

# save as pdf
dir.create(path = "runs/results_figures", recursive = T, showWarnings = F)
pdf("runs/results_figures/PCA.pdf", height = 4, width = 4)
print(pca12) # "print" only important in non-interactive use
dev.off()

# clean up
remove(pca, pca12, scores, s)

## Hierarchical Clustering
# Another nice overview plot is a heatmap of hierarchical clustering.

# we'll be using the same data as for the PCA

```



```

head(for_clust)

# before our hierarchical clustering, we need to decide on a distance metric
# type ?dist to find out which methods are available
dist(for_clust, method = "euclidean")
# more commonly we invert a correlation method, like Pearson or Spearman (?cor)
as.dist(1 - cor(t(for_clust), method = "spearman"))
# did you notice that we had to transpose the data for 'cor'?
# no, this isn't intuitive. R, is a conglomeration of a language, written by
# many different people. Some people like different defaults. Use the help
# functions, use google, find examples, keep track of how you did it last time,
# check as you go, so bugs don't propagate, hang in there.

# and we cluster, again type ?hclust to see other methods of clustering
samp_dist <- as.dist(1 - cor(t(for_clust), method = "spearman"))
gene_dist <- as.dist(1 - cor(for_clust, method = "pearson"))
hierarchy_samples <- hclust(samp_dist, method = "complete")
hierarchy_genes <- hclust(gene_dist, method = "complete")
# now our information is ready, but we cannot see it
# first we'll set up a color gradient function
blueyellow <- colorRampPalette(c("blue", "black", "yellow"))
# now let's make the heatmap
heatmap(for_clust,
  Rowv = as.dendrogram(hierarchy_samples),
  Colv = as.dendrogram(hierarchy_genes),
  labCol = NA,
  col = blueyellow(40),
  mar = c(4, 10)
)
# we'll save the image with a nifty one-liner
dev.copy2eps(file = "runs/results_figures/sampleclustering.eps")

# cleanup
remove(
  blueyellow, for_clust, gene_dist, samp_dist, hierarchy_genes,
  hierarchy_samples
)

```

7.6 Differential Expression

One of the most typical questions for an RNA-seq analysis is what is different between two samples? To answer that question we use sleuth. There is some debate about which tool is the best to detect differential expression. Here we will use Sleuth, while other possibilities are `deseq2`, `edgeR`, and `cufflinks`. Most meta analysis papers agree on `cufflinks` being less suitable. `DESeq2` and `edgeR` are both well established and work well with counts per gene and assuming a negative binomial distribution, but appear to be less appropriate for the 'estimated counts' produced by `Kallisto`. `Sleuth` is developed by the same people as `kallisto` and is supposedly most suitable for use with the results thereof. Note: In order to use `sleuth`, `kallisto` has to be run with the bootstrapping option.

The following has been adapted from this on-line tutorial: https://pachterlab.github.io/sleuth_walkthroughs/trapnell/analysis.html

```

### Analysis of differential gene expression using sleuth

library(sleuth)
library(ggplot2)
library(ggrepel)

```



```

# First we need to specify where the kallisto results are stored.
# If you didn't specify this in your kallisto script, move all kallisto results
# folders (one for each sample) by GUI or the command line into a new folder called
# "kallisto_results".

# Begin by storing the base directory of the kallisto results in a variable
base_dir <- "runs/kallisto_results/"

# Next get the list of sample IDs with
### Note: this only works, if the only folder content is the kallisto output
# (manual alternative below)
sample_id <- dir(base_dir)
sample_id

# A list of paths to the kallisto results indexed by the sample IDs is collated with
kal_dirs <- dir(base_dir, full.names = T) ## Sleuth requires full paths
kal_dirs

# Alternatively by hand:
# sample_id <- c("mock1", "mock2", "mock3", "treatment1", "treatment2", "treatment3")
# kal_dirs <- c("kallisto_results/mock1",
# "kallisto_results/mock2",
# "kallisto_results/mock3",
# "kallisto_results/treatment1",
# "kallisto_results/treatment2",
# "kallisto_results/treatment3")

# The next step is to load (here: build) an auxillary "sample to condition" / "s2c"
# table that describes the experimental design and the relationship between
# the kallisto directories and the samples:

# Double-check the order
sample_id
kal_dirs
condition <- rep(c("mock", "treatment"), each = 3)

s2c <- data.frame(sample = sample_id, condition)
s2c

# Now, we must add a column with the kallisto_directories for each sample.
# This column must be labeled 'path', otherwise sleuth will throw an error.
# The user should check whether or not the order is correct.
# In this case, the kallisto output is correctly matched with the sample identifiers.

s2c$path <- kal_dirs
s2c

# Now the "sleuth object" can be constructed.
# This requires four commands:

# (1) load the kallisto processed data into the object
so <- sleuth_prep(s2c, ~condition)

```

```

# (2) estimate parameters for the sleuth response error measurement (full) model
so <- sleuth_fit(so)

# (3) estimate parameters for the sleuth reduced model, and
so <- sleuth_fit(so, ~1, "reduced")

# (4) perform differential analysis (testing).
so <- sleuth_lrt(so, "reduced", "full")

# Now generate a table of results for analysis
treatment.vs.mock <- sleuth_results(so, "reduced:full", test_type = "lrt")

# count significant genes (e.g.)
table(treatment.vs.mock$qval <= 0.01)
# another look at the data.frame
head(treatment.vs.mock)

# <<< challenge excercises >>> #
# 1. compare the logFC edgeR calculated to that which we did
# 2. where does the difference comes from? (it's in the edgeR manual)

# now we transfer the result to our compilation data.frame 'dfr'
# actually, all we really want is the 'false discovery rate' AKA 'q_value'
dfr <- merge(dfr, treatment.vs.mock[, c("target_id", "qval")],
  by = "target_id",
  all.x = T
)

# and rename the multi-hypothesis corrected values
names(dfr)[names(dfr) == "qval"] <- "treatment.vs.mock_q_value"

# and some clean-up
# remove(sample_id, kal_dirs, condition, base_dir, s2c, so, treatment.vs.mock)

## Volcano Plot
# With this, we have the information about differential expression in the table.
# Now we can make a figure to visualize the result.
# One typical method is a volcano plot.

# we have to store the information about significance for coloring our plot

dfr[is.na(dfr$treatment.vs.mock_q_value), "treatment.vs.mock_q_value"] <- 1

dfr[dfr$treatment.vs.mock_q_value >= 0.01, "treatment.vs.mock_significant"] <- F
dfr[dfr$treatment.vs.mock_q_value < 0.01, "treatment.vs.mock_significant"] <- T

# and make a volcano plot using ggplot2
treatment.vs.mock <- ggplot(data = dfr, aes(
  x = log2FC,
  y = -log10(treatment.vs.mock_q_value),
  color = treatment.vs.mock_significant
)) +
  geom_point(size = 1, shape = 20) +

```

```

scale_color_manual(values = c(
  "FALSE" = "black",
  "TRUE" = "red"
)) +
xlab("log2 fold change") +
ylab("-log10 p-value") +
theme(legend.position = "none")

treatment.vs.mock
# go through line by line and see if you understand what is plotted
# you can also make separate plots with the first two lines, the first three
# lines, etc. to see what each line is actually doing and how it might work

# save the plot
pdf("runs/results_figures/volcano_plot_sleuth.pdf", height = 8, width = 8)
treatment.vs.mock
dev.off()

# clean up
remove(treatment.vs.mock)

# one last setup item, we will occasionally need not the transcript but the gene IDs
# the gene (locus) ID is simply the first nine characters of an AGI
locus <- substr(dfr$target_id, 1, 9)
dfr <- cbind(locus, dfr)

# we can store the data in biologist readable format by using write.table
write.table(dfr,
  file = "runs/kallisto_combined/mothertableV3.txt",
  row.names = F, sep = "\t", quote = F
)
# and we can store the data as an R object
save(dfr, file = "runs/kallisto_combined/mothertableV3.Rdata")

```

7.7 GO term enrichment

We have about 1k significantly changed genes, too many to go through by hand quickly. We now check if some functional categories are overrepresented. The idea behind this is as follows: we have 1k genes changed of 28k genes in the genome, or 3.5%. When we now look through the categories, they should all have 3.5% of their members changed if the changes were random. The statistics check whether the difference from 3.5% we see is significant. To do so, we use the package TopGO.

This analysis looks at the significantly different genes.

The biggest impediment to the use of TopGO will be to have the gene to GO term assignment in the way the package wants it. To this end, look at the file Athid2go.map in the folder. This is one way the package accepts. If you have a different species and a different way of GO term annotation, you need to reformat the file to match this one (can be done in R, but will be different for each case).

```

# as always, we need a library for that
library(topGO)

load(file = "runs/kallisto_combined/mothertableV3.Rdata")

# this reads in the GO annotation file
# you will need an equivalent file for your species!

```

```

geneID2GO_Ath <- readMappings("studies/AthalianaReferences/resources/Athid2go.map")
head(geneID2GO_Ath)
# we first prepare a factor for all genes indicating
# whether genes were upregulated "1" or not "0"
up_or_not <- dfr["log2FC"] > 0 & dfr["treatment.vs.mock_q_value"] < 0.01
up_or_not <- factor(as.integer(up_or_not))
# we want to give this list gene names that exactly match those in "Athidgo.map"
names(up_or_not) <- dfr$locus # attach gene IDs
# check
head(up_or_not)
table(up_or_not)

# we save the prepared information in an object topGO understands
# we'll focus on the ontology "BP" biological process.
# the factor marking upregulated genes goes in at allGenes
# the annotation and it's type go in at gene2GO and annot, respectively.
GOdata_sig <- new("topGOdata",
  description = "treatment.vs.mock_up",
  ontology = "BP",
  allGenes = up_or_not,
  nodeSize = 10,
  annot = annFUN.gene2GO,
  gene2GO = geneID2GO_Ath
)

# now we need to do the statistical test, classic Fishers Exact Test is chosen
resultsGOfisher <- runTest(GOdata_sig, algorithm = "classic", statistic = "fisher")
# this extracts a sorted, summary table
tableGOresults <- GenTable(GOdata_sig,
  classicFisher = resultsGOfisher,
  topNodes = length(resultsGOfisher@score)
)

# the P-value is in the column "classicFisher". We will also calculate the q_value.
tableGOresults$q_value <- p.adjust(tableGOresults$classicFisher, method = "BY")
# BY is a different method for calculating FDR, that's more dependency tolerant.
# filter to significant
tableGOresults <- tableGOresults[tableGOresults$q_value < 0.05, ]

# people always ask which genes are behind the GO terms
# topGO provides a function to find them
genesInTerm(GOdata_sig, "GO:0050896")
# now let's run this for all our top terms
# broken down by whether they were significantly up or not
# we pre-cache a character vector of the significant gene IDs
sig_genes <- dfr$locus[dfr["log2FC"] > 0 & dfr["treatment.vs.mock_q_value"] < 0.01]
# we'll save a mini-function to get and organize the IDs
genes_in_term_by_sig <- function(ontology, whichGO, sig_genes) {
  all_ids <- genesInTerm(ontology, whichGO)[[1]]
  # break up by significance
  by_sig_ids <- split(all_ids, all_ids %in% sig_genes)
  # vector to comma-separated string
  by_sig_ids <- sapply(by_sig_ids, paste, collapse = ",")
  names(by_sig_ids) <- c("ns_ids", "sig_ids") # set names
  return(by_sig_ids)
}
# test the function

```

```

genes_in_term_by_sig(GOdata_sig, "GO:0050896", sig_genes)
# use the function
tableGOresults <- cbind(
  tableGOresults,
  t(sapply(
    tableGOresults$GO.ID,
    function(x) genes_in_term_by_sig(GOdata_sig, x, sig_genes)
  ))
)
tail(tableGOresults) # tail only because the gene lists were shorter

# our final steps are to export the table in biologist readable format
write.table(tableGOresults,
  file = "runs/results_figures/GO_treatment_vs_mock_up_Fisher.txt", sep = "\t",
  row.names = FALSE
)

# and to export a graphical representation
# if you want to see more GO terms than 20, change firstSigNodes
printGraph(GOdata_sig, resultsGOfisher,
  firstSigNodes = 20,
  fn.prefix = "runs/results_figures/GO_treatment_vs_mock_up",
  useInfo = "all", pdfSW = TRUE
)

# <<< challenge assignments >>> #
# 1. perform GO enrichment on down-regulated genes

# clean up
remove(
  geneID2GO_Ath, GOdata_sig, resultsGOfisher, sig_genes, tableGOresults,
  up_or_not
)

```

GO term lists are notoriously difficult to show in nice figures and also difficult to interpret. Generally, you want many similar terms up to be convinced something is real. Revigo (on-line tool) can help you to summarize the GO terms but essentially you need your biologist knowledge to understand them.

We chose a rather simple example. Among the terms enriched in the upregulated genes you can always observe the words defense and immune. You have to know that systemic acquired resistance also refers to the immune system, as does response to biotic stimulus. Clearly, the treatment induces the defense response.

Among the enriched terms in down-regulated genes appears photosynthesis, which occurs in the plastid, and involves glyceraldehyde-3-phosphate, pigments and NADP. So clearly, although the names are different, what they describe is similar. There is no way but simply knowing this. There are thousands of GO terms described. The figures which were also produced are sometimes helpful. The GO terms connected by lines are the ones defined as in parent:daughter relationships.

7.8 MapMan

MapMan is an alternative way of looking at the overall patterns. Usually, you load all genes with their fold-changes, not just the significant ones. In a perfect world, MapMan and GO should give you similar results. One caveat is that the annotation behind GO and MapMan was made by different people and therefore different areas of plant biology are covered differently.

Before starting with MapMan we will be making use of the size of the class to install MapMan in parallel on all course computers. MapMan's provided install script can be found both on sciebo, and in the

Raumlaufwerke folder. Run the following with java, and follow the instructions of the installation wizard.

```
java -jar </path/to/>/MapManInst-3_6_0RC1.jar
```

We only need to export our fold-change table from R. Again, we have to make sure that our gene identifiers match the ones used in the MapMan tool.

Open MapMan, click on mappings and look at the Ath_AGI mapping. Click on until you see AGI codes and observe how they are formatted—again, no transcript suffix, only the locus. Back to R.

```
load(file = "runs/kallisto_combined/mothertableV3.Rdata")

# now we make a data.frame with the data required for Mapman loading
# if you have more fold-changes, you can load more than one
forMapman <- dfr[, c("locus", "log2FC")]
head(forMapman)
# now we export the data.frame in biologist and mapman readable format
dir.create(path = "runs/mapman", recursive = T, showWarnings = F)
write.table(forMapman,
  file = "runs/mapman/forMapmanloading.txt",
  quote = F, sep = "\t", row.names = F
)
remove(forMapman)
```

Now back to MapMan. Right click on Experiments, choose add data. Select your exported table named forMapmanloading.txt. Now you can choose if there is a header (yes!), whether you have decimal point or decimal comma, and so on and so forth. The default works for us. Click okay

Now click on Pathways, on Overview and choose Metabolism_overview by double click. You will have to choose a mapping; we use Ath_AGI. Click okay.

If nothing is displayed, click on your file shown in Experiments. I always change the color code to +red and-blue and the scale to whatever fits my dataset (for this one I'd use 3).

Clearly, photosynthesis, the CBB cycle and photorespiration are down. If you would like to test that statistically, Mapman does that for you. Look at the stuff below the figure and pull it up. You will see the Wilcoxon Rank Sum Test scores. Correct the p-values for multiple hypothesis testing and sort the table by probability. Some of the significantly changed pathways we can see, most are not displayed. Look at the other options for visualization to see, if anyone fits your needs.

You can make your own pathway figures and map the MapMan bins onto them. Refer to the MapMan manual for that!

7.9 Adding information from public data sources

Combining information from multiple sources is often helpful, or even necessary to fully understand one's results. This becomes harder to write a standardized protocol for, however, as one researcher might simply be interested in including the TAIR annotation with their "mother table" for ease of looking through the gene descriptions, and another might be interested in knowing whether there's a significant overlap between their study and a particular paper. That said, there is frequently a similar line of attack. First, get the desired comparative data in an organized format such as a list of gene IDs or a table (csv/tsv/xls) with gene IDs and associated values. Second, make sure the gene IDs are comparable (are they from the same genome/annotation release, do you have transcript IDs when you wanted gene IDs, are they both upper/lower case). Third, combine data with yours (e.g. merge). Finally, in many cases you will visually and statistically evaluate whether there is an overlap or correlation between the studies.

7.9.1 Including gene descriptions from TAIR

First step first, we need to get the descriptions, and not by copying them one by one from the website. Most biological databases have a bulk download page if you look, and TAIR is on the easy side. From <https://arabidopsis.org> you simply need to go to Download:Genes, select the annotation (TAIR10), and you already have a list of what is available for download. We'll take "TAIR10_functional_descriptions", which is a tab-separated text file. Save, copy or move this file to studies/AthalianaReferences/resources for ease of access. The ID format matches, so adding the descriptions to our major data.frame in R is very simple.

```
load(file = "runs/kallisto_combined/mothertableV3.Rdata")

# import the tair annotations
tair_anno <- read.delim(
  "studies/AthalianaReferences/resources/TAIR10_functional_descriptions.txt"
)
# merge with main table
dfr_anno <- merge(
  x = dfr, y = tair_anno, by.x = "target_id",
  by.y = "Model_name", all.x = TRUE
)
# we can now search these descriptions
i_pathogen <- grep("pathogen", dfr_anno$Computational_description, ignore.case = T)
i_pathogen # list of indexes with "pathogen" in description
# look at first hit
dfr_anno[i_pathogen[1], ]

# clean up time
remove(dfr_anno, tair_anno, i_pathogen)
```

7.9.2 Including MapMan annotations

Frequently you want to have more flexibility working with a dataset than is possible with the provided GUI programs. We'll now import the MapMan annotations into R, and look briefly at what else one could do with them. For a handful of plant species, MapMan annotations are provided in the 'store' at <https://mapman.gabipd.org/web/guest/mapmanstore>. More commonly, they can be created for a species using the Mercator webtool <http://www.plabipd.de/portal/web/guest/mercator-sequence-annotation>. Assigning MapMan annotations with Mercator normally runs in less than 15min and avoids any annotation version trouble. For today, the results are provided in the file mapmanTAIR10.txt.

```
load(file = "runs/kallisto_combined/mothertableV3.Rdata")

library("reshape2") # this library gives us the "melt" function
# basically, makes preparing data for ggplot2 easier
# import
mapman <- read.delim("studies/AthalianaReferences/resources/mapmanTAIR10.txt")
head(mapman)
# we can remove the quotations now
mapman <- data.frame(apply(mapman, c(1, 2), function(x) gsub("'", "", x)))
# Hmmmm, this time the IDs don't match, clean up time!
# in the "dfr" table we can use the locus ID in uppercase
head(dfr$locus)
# in the MapMan data we have gene IDs in lowercase
head(mapman$IDENTIFIER)
# changing to upper case is easy
mapman$IDENTIFIER <- toupper(mapman$IDENTIFIER)
# then we merge the tables
```

```

dfr_mapman <- merge(
  x = dfr, y = mapman, by.x = "locus",
  by.y = "IDENTIFIER", all.x = TRUE
)

## Now we can look at the fold change of differentially
## expressed transcription factors
dfr_mapman_sig <- dfr_mapman[dfr_mapman$treatment.vs.mock_significant, ]
i_TFs <- grep("^RNA.regulation of transcription", dfr_mapman_sig$NAME)
head(dfr_mapman_sig[i_TFs, c("log2FC", "NAME")])

## or we can visualize redox response ##
i_redox <- grep("^redox", dfr_mapman_sig$NAME)
# get significant redox gene tpm
to_plot <- dfr_mapman_sig[i_redox, grep("tpm", colnames(dfr_mapman_sig))]
colnames(to_plot) <- gsub("_tpm", "", colnames(to_plot)) # fix names
to_plot <- to_plot[apply(to_plot, 1, max) > 100, ] # filter to reduce noise
# transform to Z-score (mean center, divide by standard deviation)
to_plot <- t(scale(t(to_plot)))
head(to_plot)
# melt, helps us prepare data for ggplot2
to_plot <- melt(to_plot)
head(to_plot)
# now more remain, just for our own organization this time
colnames(to_plot) <- c("gene_index", "replicate", "z.score")
redox_plot <- ggplot(data = to_plot, aes(x = replicate, y = z.score, group = gene_index)) +
  geom_line()

redox_plot
dev.copy2eps(file = "runs/results_figures/redox.eps", width = 6, height = 4)
# notice anything interesting about the replicates?

# <<< challenge assignment >>> #
# 1. we could have imported the mapman table and removed quotes in
# one step. Do so (hint ?read.delim).
# 2. Take a careful look at the results and original file, and
# and try and troubleshoot what went wrong.

save(mapman, dfr_mapman, dfr_mapman_sig, file = "runs/mapman/mapman.RData")

# clean up time
remove(mapman, dfr_mapman, dfr_mapman_sig, i_redox, i_TFs, to_plot, redox_plot)

```

You can, of course, automate a Fisher's exact test or Wilcoxon test for every category in R, visualize many samples at once by different bin levels, or whatever suits your purposes.

7.9.3 Special list - from a review paper

Frequently, whole genome annotations have not been updated with the very latest information or you are interested in a more obscure list, so you want to compare data with another paper directly. This often makes the first step of getting the data in an organized table harder. Let's look at the review paper <https://doi.org/10.1007/s11427-016-0048-4> "Diverse roles of SERK family genes in plant growth, development and defense response". In table 1, it has a list of SERK interacting genes, which we might be interested in, but they are only available in pdf format. There are many ways to approach this. You can for instance copy the whole table, and clean up any formatting issues by hand. Often you can get the formatting to work by pasting it into either a spreadsheet program or a text file. However, as we've

worked with *A. thaliana* a lot before, we already had a handy script to find all of the AGIs out of a text file and will use this.

First, copy all of table 1 (or even the whole paper) into a text file and save it as `serk_interacting.txt`

```
# now run the following in bash
workflows/agi_finder.py -i studies/Fan2016_SERK_review/resources/serk_interacting.txt \
> studies/Fan2016_SERK_review/resources/serk_interacting.agi
less studies/Fan2016_SERK_review/resources/serk_interacting.agi
```

For today, we only needed the AGIs and not the additional information, so this will do.

```
load(file = "runs/kallisto_combined/mothertableV3.Rdata")

# moving back to R, import the data
serk <- read.csv("studies/Fan2016_SERK_review/resources/serk_interacting.agi", header = F)
# make another copy of the main table
dfr_serk <- dfr
# setup a TRUE/FALSE column for SERK interaction
dfr_serk$serk_interacting <- FALSE
dfr_serk$serk_interacting[dfr_serk$locus %in% serk[, 1]] <- TRUE
# this makes for easy subsetting (if not strictly necessary for this example)
# visualize, for example, the logFC of SERK interacting genes.
to_plot <- dfr_serk[dfr_serk$serk_interacting, "log2FC", drop = F]
px <- ggplot(to_plot, aes(log2FC)) +
  geom_histogram(binwidth = 0.5) +
  xlim(-max(abs(to_plot)), max(abs(to_plot)))
px
# hmm, it looks like these genes are upregulated
# more often than not, let's test it
wilcox.test(dfr_serk$log2FC ~ dfr_serk$serk_interacting)
# looks like there's a difference, whether something like this
# is interesting or not will depend heavily
# on the biological knowledge you're bringing in to the question.

# clean up time
remove(serk, dfr_serk, to_plot, px)
```

OK, we'll never be able to cover all possible comparative data and data formats you might want to look at, but hopefully you have some ideas on where to start now. Take a look around, if you are ahead of your neighbors this would be a great time to try importing contextual data you would actually be interested in for your studies.

7.10 Further Clustering

7.10.1 K-means

```
library(RColorBrewer)
library(gridExtra)

# We've looked at a couple of clustering methods already (PCA, Hierarchical)
# Now we'll look at k-means.
# Advantages: fast
# It depends: favors even cluster sizes
# Challenges: user-defined 'k' for number of clusters, non-deterministic

# This only makes sense on a somewhat larger dataset
to_cluster <- read.csv("studies/kmeansexample/resources/to_cluster.csv",
```

```

    row.names = 1
  )
  head(to_cluster)
  dim(to_cluster)
  # with a larger dataset like this,
  # it's often appropriate to average replicates before clustering
  groups <- factor(gsub("\\.[1-9]", "", colnames(to_cluster)))
  mean_to_cluster <- sapply(
    levels(groups),
    function(x) rowMeans(to_cluster[, which(groups == x)])
  )
  head(mean_to_cluster) # we'll need new column names now
  colnames(mean_to_cluster) <- levels(groups)

  # filter, log transform, and convert to z-scores, much like we've done before
  sub_to_cluster <- mean_to_cluster[apply(
    mean_to_cluster, 1,
    function(x) min(x) > 0 & max(x) > 50
  ), ]
  sub_to_cluster <- log2(sub_to_cluster)
  sub_to_cluster <- t(scale(t(sub_to_cluster)))
  # check
  head(sub_to_cluster)
  dim(sub_to_cluster)
  # run kmeans
  km <- kmeans(sub_to_cluster, 8)
  str(km) # take a look at the kmeans object
  km$centers # these are the 'center' of each produced cluster
  head(km$cluster) # which cluster each gene is in
  km$tot.withinss # summed up sum-of-squares between each gene and cluster center
  ?kmeans # more information (particularly under Value)

  # OK, so far so good, but why 8 clusters? Let's choose more carefully
  # we can use tot.withinss as a quality measure, and just try different values
  k_to_test <- 2:42
  km_k2up <- sapply(
    k_to_test,
    function(x) kmeans(sub_to_cluster, x, nstart = 3)$tot.withinss
  )
  # we'll already start getting ready for visualization
  to_plot <- data.frame(ss = km_k2up, k = k_to_test, run = "real")
  # Now we'll want negative controls, which shouldn't cluster well
  n_negative_controls <- 7
  for (i in 1:n_negative_controls) {
    scrambled_to_cluster <- t(apply(sub_to_cluster, 1, sample))
    km_k2tok24negative <- sapply(
      k_to_test,
      function(x) kmeans(scrambled_to_cluster, x)$tot.withinss
    )
    to_plot <- rbind(to_plot, data.frame(
      ss = km_k2tok24negative,
      k = k_to_test,
      run = paste0("neg", i)
    ))
  }
  # can you think of a reason that there are more warnings with the scrambled data?

```

```

warnings()
# now let's get this ready for visualization
choose_k <- ggplot(data = to_plot, aes(x = k, y = ss, color = run, group = run)) +
  geom_line() +
  geom_point(size = 1, shape = 20) +
  scale_color_manual(values = brewer.pal(length(unique(to_plot$run)), "Set1"))
choose_k
# we can either use the 'elbow rule' (peak of the curve is around 8)
# or we can take the maximum difference between real and scrambled data
neg_minus_real <- sapply(
  paste0("neg", 1:n_negative_controls),
  function(x) to_plot$ss[to_plot$run == x]
)
neg_minus_real <- rowMeans(neg_minus_real) - to_plot$ss[to_plot$run == "real"]
# neg_minus_real <- to_plot[to_plot$run=="neg1", "ss"] -
# to_plot[to_plot$run=="real", "ss"]
neg_minus_real <- data.frame(neg_minus_real, k = k_to_test)

choose_k_by_diff <- ggplot(neg_minus_real, aes(y = neg_minus_real, x = k)) +
  geom_bar(stat = "identity")
choose_k_by_diff
# and if it's still hard to see where greatest differences is
neg_minus_real[order(neg_minus_real$neg_minus_real, decreasing = T)[1:5], ]
# you may notice, if you run this again, that there are slight variations
# we recommend looking at the graphs, as the eye is hard to fool
# keep in mind there is no 'correct' k,
# you might choose fewer for easier visualization
# or more for identifying gene sets for CRE prediction, etc...
# we'll continue with 6, and run the clustering 100 more times, taking the best
km <- kmeans(sub_to_cluster, 6, nstart = 100)
# let's get the data from one cluster
a_cluster <- sub_to_cluster[km$cluster == 1, ]
a_cluster <- melt(a_cluster)
head(a_cluster) # those column names are hard to remember
colnames(a_cluster) <- c("locus", "sample", "z.score")

a_cluster_plot <- ggplot(
  data = a_cluster,
  aes(x = sample, y = z.score, group = locus)
) +
  geom_line(alpha = 0.1, color = "blue3") +
  labs(x = "", y = "z-score") +
  theme(
    text = element_text(size = 20),
    axis.text.x = element_text(angle = 90, vjust = 1)
  )

# and now run that for every cluster and save it
cluster_plots <- list()
for (i in 1:6) {
  a_cluster <- sub_to_cluster[km$cluster == i, ]
  a_cluster <- melt(a_cluster)
  colnames(a_cluster) <- c("locus", "sample", "z.score")

  a_cluster_plot <- ggplot(
    data = a_cluster,

```

```

aes(x = sample, y = z.score, group = locus)
) +
geom_line(alpha = 0.1, color = brewer.pal(6, "Set1")[i]) +
labs(x = "", y = "z-score") +
theme_bw() +
theme(
  text = element_text(size = 20),
  axis.text.x = element_text(angle = 90, vjust = 1)
) +
theme(aspect.ratio = 1) +
ggtitle(paste("Cluster", i))

cluster_plots[[i]] <- a_cluster_plot
}

pdf("runs/results_figures/kmeans.pdf", width = 10, height = 16)
do.call(grid.arrange, cluster_plots)
dev.off()

```

7.10.2 Heatmaps - gene set

We've seen heatmaps at a large scale when displaying the hierarchical clustering above. Sometimes however, they are useful just for looking at dozens of genes at once. Let's look at the MapMan category, stress.biotic.signalling as an example.

```

load("runs/mapman/mapman.RData")

# get indices of biotic stress signalling genes
i_biotic <- grep("stress.biotic.signalling", dfr_mapman$NAME)
# get tpm subtable for plotting
to_plot <- dfr_mapman[i_biotic, grep("tpm", colnames(dfr_mapman))]
to_plot <- log2(to_plot + 1)
colnames(to_plot) <- gsub("_tpm", "", colnames(to_plot))
# the melt function likes names (here AGIs)
to_plot$Name <- dfr_mapman[i_biotic, "locus"]
# make a reasonable order
# (here sorted by mean, but clustering would work too)
to_plot <- to_plot[order(rowMeans(to_plot[, 1:6])), ]
# this will force the plot to keep the current ordering
to_plot$Name <- factor(to_plot$Name, levels = unique(to_plot$Name))
# setup data.frame ggplot2 style
x <- melt(to_plot)
colnames(x) <- c("AGI", "Replicate", "log2")
# make plot
px <- ggplot(x, aes(Replicate, AGI)) +
  geom_tile(aes(fill = log2)) +
  scale_fill_gradient(low = "black", high = "red")
# display plot
px
dev.copy2eps(file = "runs/results_figures/biotic_signaling.eps")

```

8 Long Read Sequencing

Important qualifier: while we've worked a lot with RNAseq, we've so far only had our own projects working with long reads of DNA (and more Nanopore than PacBio). So writing this section was learning by doing for us, too. We'd rather look forward than look back, but please take everything with a grain of salt / don't think that the following represents stable repeatedly-tested pipelines.

Recent advances in long read sequencing are making drastic changes to what is possible in terms of RNA sequencing. While we've covered the technologies more thoroughly in a lecture, here is a drastic over simplification.

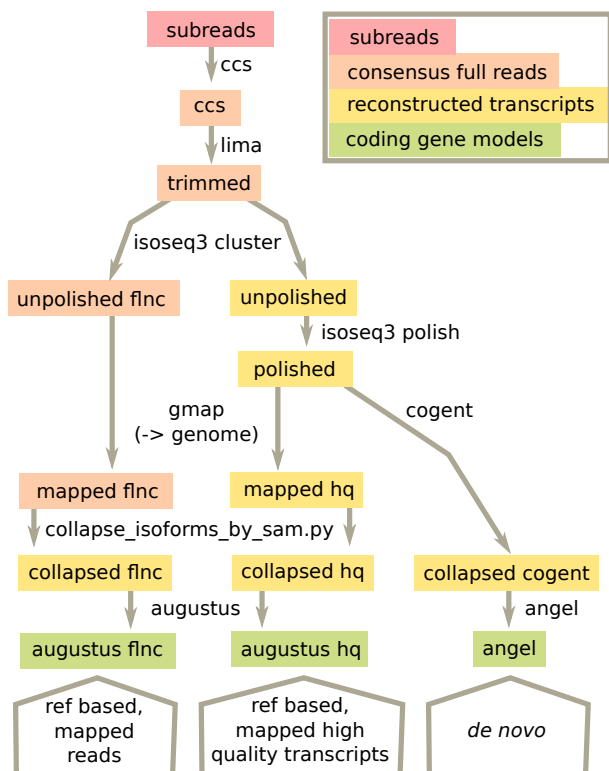
Pacific Bioscience's Iso-Seq provides a way to repeatedly sequence potentially full length cDNA of transcripts and their reverse complement, and the sequencing will proceed in a circle until the polymerase falls off. Thus the preliminary base called read produced requires more processing than some basic trimming, but can have very high quality after consensus calling.

Oxford Nanopore Technology's direct RNAseq measures the current across a membrane while the RNA molecule itself (and in some cases it's single cDNA strand) proceeds through a pore across the membrane. While the error rate is high, the output appears better than PacBio, and there are some exciting possibilities in detecting RNA modifications.

8.1 The Plan

While the throughput of these technologies is improving, we would argue that as the sheer read count is *currently* (at least) an order of magnitude lower than for Illumina, if the goal is simply quantification, use Illumina. That said, if the goal is establishing coding gene models (with or without a reference) or something more specific, like quantifying alternative splicing, the long read technologies can be extremely helpful. Here we'll look at three options for reconstructing coding gene models with example Iso-Seq data. As with the Illumina RNAseq analyses exactly which analysis one chooses will depend upon the exact data one has and what sort of reference is available.

In the left-most option, we'll map the preprocessed reads directly to a reference, collapse them to transcripts, and use the transcripts to support gene calling. This is also the most similar to a tractable approach for ONT direct RNAseq data. In the middle option, we will cluster the preprocessed reads into draft transcripts and then map these to the reference genome and perform gene calling as in the first option. In the right-most option, we will look at a pipeline to make gene models without any reference or with a very poor reference.



8.2 Resources

Many of the commands are based on the following resources, and even if I were to repeat this on similar data in just three months I would check back here to see if anything major has changed.

The main Iso-Seq pipeline (trimming-polishing):

Main organizational page:

https://github.com/PacificBiosciences/IsoSeq_SA3nUP/wiki

In particular:

[isoseq3](#)

[Tutorial:-Installing-and-Running-Iso-Seq-3-using-Conda](#)

[Cupcake ToFU: supporting scripts for Iso Seq after clustering step](#)

The *de novo* options:

[Running-Cogent](#)

[Angel](#)

8.3 Data description

You can find the data for this section here:

```
ls assays/Zhu2017_IsoSeq/dataset
```

This is a fairly early Iso-Seq dataset designed to investigate any role of alternative splicing in the abscisic acid response in Arabidopsis seedlings (<https://www.ncbi.nlm.nih.gov/pubmed/28407323>). This study also relied heavily on Illumina data, which we won't be using here. Just a brief caveat that as Iso-Seq develops (and becomes cheaper) much deeper sequencing and more replicates will almost certainly be expected than is seen here. But it's plenty for the workshop; we'll even only be using one of their two conditions.

While the raw .hd5 files were downloaded from SRA, we've done some of the initial pre-processing for you a) for the sake of time and b) because the early steps in particular are very specific to a given technology and can often be provided by a sequencing center.

Still, it looked like this:

```
# convert from Hierarchical Data Format files to bam
# (sequel anyways directly outputs bam for newer data)
bax2bam raw/m16*bax.h5
# organize things by moving output into it's own directory
mkdir -p runs/isoseq/subreads runs/isoseq/ccs
mv m16* subreads/
# take the raw subreads and convert them to the 'circular consensus sequence'
bam=m161031_124550_42199_c100941222550000001823217706101620_s1_X0.subreads.bam
ccs --numThreads=4 --noPolish --minLength=50 \
    --maxLength=15000 --minPasses=1 --minPredictedAccuracy=0.8 \
    --reportFile=runs/isoseq/ccs/keep.ccs.report --minSnr=3.75 --minZScore=-999 \
    --maxDropFraction=0.8 runs/isoseq/subreads/$bam runs/isoseq/ccs/m16.ccs.bam
# we also truncated the output file name here for the workshop, although
# we'd generally encourage you too keep the naming as consistent as possible
```

8.4 First Look

So first thing first, take a look at the data.

```
# check that you're in the IsoSeqData directory and
ls -R runs/isoseq/
# OK, this is new, as you can see there are neither fasta nor fastq sequencing files
# Instead the .bam files are the main sequence containers,
# we've seen .sam/.bam files before for an _alignment_, hmmm...
# further, the following won't get us very far
less runs/isoseq/subreads/m16*.subreads.bam
```

```
# note that throughout this whole section using the full file name instead
# of m16* is always OK. Just hit tab.
```

So what's going on here? While they aren't aligned to a *reference*, one of PacBio's raw reads is well represented aligned back to *itself* as the read is basically going in circles. The circular consensus sequence is no longer aligned to anything, but it avoids a trivial format conversion. It also allows a little more flexibility for PacBio to sneak some additional data in.

We're going to need some ways to view .bam files now. They have a text-based (and much more hard-drive-hungry) sister format, .sam. Luckily interconversion is easy.

```
# convert to the text format, and pipe the first few lines to a file
# so we can look at them in more detail
samtools view runs/isoseq/subreads/m16*.bam|head > runs/isoseq/subreads/head_subreads.sam
samtools view runs/isoseq/ccs/m16*.bam|head > runs/isoseq/ccs/head_ccs.sam
```

Now take a look at the files with either `less` or `gedit`. We'd also encourage you to use `cut -f N <file>` to look just at a specific column (N) when appropriate. There are a few things to notice.

- The basic format: google 'sam format' or checkout <https://samtools.github.io/hts-specs/SAMv1.pdf> for a full description, but the basic idea of the format is that it has different columns for: the sequence, its quality information, where and how it maps to a reference (or not), and some.
- OK, it's not quite normal .bam/.sam format, PacBio has expanded it some, if you're curious see: <https://pacbiofileformats.readthedocs.io/en/3.0/BAM.html>
- Let's look at the sequence itself (column 10), note the large stretches of AAAA's and TTTT's, these are your poly-A tail being read as the circle turns round.
- Judging just off of the poly-A tail, do you notice the quality difference between the subreads and the ccs?
- Look at the sequence identifiers in column 1. For the subreads you see that after taking the top few lines with `head` we only actually see sub reads from one read (.../8/) while sub-coordinates are given in the final field. In contrast we see different reads in the ccs and the final field just says 'ccs'.
- Can you find the primer sequences (see `clonetechnology.SMARTer.fa`) in the reads? Hint: you can orient where to look with the poly-A tail.

Unfortunately there really is currently no equivalent of FastQC or any at-the-start quality control for PacBio data. The quality can be estimated by mapping to a reference, but we're getting there in good time anyways; so for now just try and get a feel for the sort of sequence we're looking at. The subreads go in circles, the ccs is the consensus of the subreads from individual full-reads. Both still have adapters and poly-A tails.

8.5 Trimming, Clustering and Polishing

8.5.1 Leveling up relative to day one

On the first day with the Illumina RNAseq data, we always just put results in our main directory, and it was pretty full. While this made things a bit easier at the beginning, it becomes very confusing as a project grows. Both because today's work has the tendency to produce multiple and complicated output files and because it is generally better form, we will try and use sub directories to structure the project a bit more.

Similarly, on the first day we wrote out, at least for the Kallisto pipeline, almost every parameter. Today we'll just take the basic explanations from each tool's usage function, and only stop to explain parameters that are particularly confusing or where we've deviated from the 'standard analysis' pipeline. But please feel free to ask questions (or point out anything confusing)!

8.5.2 Adapter trimming

We'll trim off the adapters with `lima`. If the run contained multiple multiplexed libraries, we could also perform demultiplexing with `lima`.

```
mkdir -p runs/iseq/trimmed/
lima runs/iseq/ccs/m16.ccs.bam workflows/pacbiosciences_lima/clonetech_SMARter.fa \
  runs/iseq/trimmed/m16.ccs.bam --no-pbi --iseq
# The first three arguments are obvious from the usage
# Usage: lima [options] INPUT BARCODES OUTPUT
# --no-pbi since we won't need the .pbi output
# --iseq is necessary to tell it what to expect for the primers

# let's check the output
ls runs/iseq/trimmed/
# in particular, the summary file might be helpful
less runs/iseq/trimmed/m16.ccs.lima.summary
# In addition to removing adapters
# we see lima removed any reads with mismatched adapters
# and now let's convert some of the .bam output to .sam as above
samtools view runs/iseq/trimmed/m16.ccs.primers_5p--primers_3p.bam | head \
  > runs/iseq/trimmed/head_trimmed.sam
less runs/iseq/trimmed/head_trimmed.sam
# notice that _most_ of the adapters are now gone and that
# _most_ of the poly-A tails are at the end of the sequence now
```

8.5.3 Clustering

So at this point we have some reads that are *in some ways* similar to that which we had after running Trimmomatic on the Illumina data. But not quite. We just saw the remaining poly-As and adapters that were in the middle of some reads, which may be caused by concatemers. Further, as the reads have a poly-A in them, we essentially have strand information.

The next step, `iseq3 cluster`, cleans up any concatemers and orients reads while clustering different reads that *appear to* represent the same transcript.

Note that the primary goal of this step is the reconstruction of the gene models, which should be done with all data, not sample by sample. So if we were using more samples, we'd want to run `dataset create` first, which would link files in a way that they could all be processed together.

```
mkdir -p runs/iseq/unpolished/
# from the help function:
# iseq3 cluster [options] input output
iseq3 cluster runs/iseq/trimmed/m16.ccs.primers_5p--primers_3p.bam \
  runs/iseq/unpolished/m16.unpolished.bam --verbose --require-polya
# again, we have a lot of output
ls -sh runs/iseq/unpolished/
# there's two files we really care about, and both are .bam
# *.unpolished.flnc.bam contains the fully cleaned, AKA
# "full-length non-chimeric" reads. Finally.
# *.unpolished.bam has draft reconstructed transcripts

# we can also get a report on how many flnc reads were used
# for each draft transcript
iseq3 summarize runs/iseq/unpolished/m16.unpolished.bam \
  runs/iseq/unpolished/m16.summary.csv
less runs/iseq/unpolished/m16.summary.csv
```


After this, the pipelines we're looking at today start to diverge, with the first "ref based, mapped reads" option continuing with the flnc reads and the others continuing to polishing, below.

Optional challenge: We got to this step, compared it to the reported results from the paper to see if the basics (e.g. number of transcripts produced) seemed similar, and freaked out a bit.

```
# how many draft reconstructed transcripts did we get?
samtools view runs/isoseq/unpolished/m16.unpolished.bam|wc --lines
```

Can you see why we were worried? Can you also figure out why we decided the difference was nothing to worry about, just a reason to read papers with care?

8.5.4 Polishing

In terms of quantity, each draft transcript was constructed from an average of around 10 full-length non-chimeric reads, which sounds OK at first. but if you glanced at `runs/isoseq/unpolished/m16.summary.csv` or maybe even made a histogram, you know that they are definitely not evenly distributed. Indeed, well over half are constructed from just 2-3 flnc reads. But at the very start of this pipeline we had a lot more data, right? The subreads.

The next step is to take the draft 'unpolished' transcripts and carefully polish them with all that subread data we had at the very very start. This will allow any reads that were dropped (e.g. because they were not full length) to still contribute to the quality of the final sequences, and further allow for a high quality ccs read from 10+ subreads to essentially be trusted more than a ccs read with just a single subread.

```
mkdir -p runs/isoseq/polished/
# isoseq3 polish -h # uncomment for explanation
isoseq3 polish runs/isoseq/unpolished/m16.unpolished.bam \
  runs/isoseq/subreads/m16*.subreads.bam \
  runs/isoseq/polished/m16.polished.bam
```

OK, this step is going to take a long time (maybe an hour?). It's working with 5GB of reads after all. But we actually want to be able to compare the pipelines in the end, so we didn't want to unnecessarily subset this data.

In the mean time we can continue with the "reference based, mapped reads" / "left most" option.

If something goes wrong or if you get through the "Mapping FLNC reads" part before it finishes. There is a copy of the expected output at `runs/_BackUpData/isoseq/polished/`. Feel free to copy the data from there if need be.

```
cp runs/_BackUpData/isoseq/polished/* runs/isoseq/polished/
```

8.6 Coding Genome Definition

8.6.1 Reference Based

Mapping FLNC reads. While polishing is running happily, we're going to check out what happens when we take the cleaned up "full-length non-chimeric consensus reads" and map them to the genome directly. This method will depend upon the genome to help clean up any errors in the reads, but let's face it, *sometimes* you probably should trust the genome over the new transcriptome. Not always of course, but sometimes, often even.

GMAP is a pretty fast way to map full transcripts or similar back to the genome. But like most aligners, it expects .fasta or .fastq files.

Open a new terminal (Ctrl+Alt+T) so you can work while polishing runs.

```
# activate the virtual environment in the new terminal
source $HOME/Documents/venv/bin/activate
```

Now let's take a look at the 'flnc' reads

```
# we have a .bam file for the flnc reads
samtools view runs/isoseq/unpolished/m16.unpolished.flnc.bam | \
  head > runs/isoseq/unpolished/head_flnc.sam
less runs/isoseq/unpolished/head_flnc.sam
# two things:
# First, we finally have some cleaned-up mappable looking reads
# Second, the data from a fasta file is just a subset of what we see
# the read name (column 1), and the sequence (column 10)
# we're just going to make a fasta file out of the bam with simple text manipulation
samtools view runs/isoseq/unpolished/m16.unpolished.flnc.bam | \
  awk '{print ">"$1"\n"$10}' > \
  runs/isoseq/unpolished/m16.unpolished.flnc.fa
# we simply googled 'bam to fasta awk', just FYI
head runs/isoseq/unpolished/m16.unpolished.flnc.fa
```

Back to GMAP, like the other aligners we've seen, GMAP is going to need an index, and then we can map to this.

```
# make index (-D where -d index_name input.fa)
gmap_build -D runs/gmap_index -d Athaliana \
  studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa
# map reads
mkdir -p runs/isoseq/mapped/
# this should take about 8 min with one thread
# add -t N to speed it up if polishing is finished
gmap -D runs/gmap_index -d Athaliana -f samse -n 0 --cross-species \
  --max-intronlength-ends 200000 -z sense_force \
  runs/isoseq/unpolished/m16.unpolished.flnc.fa 2> \
  runs/isoseq/mapped/m16.flnc.gmap.log > runs/isoseq/mapped/m16.flnc.sam
# the parameters are simply those recommended for isoseq, but briefly
# -D, -d : to find the index made above
# -f samse : to export .sam format
# -n 0 : allow chimeric alignments
# --max-intronlength-ends 200000 : more realistic max for EUK introns
# -z sense_force : since sequences were oriented 5'-3' by poly-A tail
# 2> runs/isoseq/mapped/m16.flnc.gmap.log : redirect copious standard error output to file
# > runs/isoseq/mapped/m16.flnc.sam : finally redirect standard out to our sam file

# now we have a "normal" sam/bam file
less runs/isoseq/mapped/m16.flnc.sam
# note: in columns 3, and 4 we have the chromosome and position of mapping

# we'll want a sorted bam file for the next step
samtools sort runs/isoseq/mapped/m16.flnc.sam > runs/isoseq/mapped/m16.flnc.sorted.bam
```

OK, we have mapped data, it's time to take a break and really look at our long read data, and also how it compares to the Illumina reads we had before. Feel free to pair up with your neighbor for the next part and have one person run the Illumina half and the other the Iso-Seq half.

```
# first to visualize the reads we'll need indexes
# one for the genome
samtools faidx studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa
# one for the new mapping of flnc reads
```

```
samtools index runs/iseq/mapped/m16.flnc.sorted.bam
# for the Illumina reads we need both to convert to bam and sort
samtools sort runs/hisat_results/treatment1.hisat.sam \
> runs/iseq/mapped/illumina.sorted.bam
# and we need an index
samtools index runs/iseq/mapped/illumina.sorted.bam
```

To look at the assembly, we want to briefly install a genome browser, Tablet, *on the host machine*.

Open a new terminal on the host machine, and run

```
cd $HOME/rnaseq-workshop/
./workflows/tablet_linux_x64_1_21_02_08.sh
# click through the installer, it should open a
# GUI genome visualization tool when done.
```

In Tablet, you'll want to click to "Import an assembly". Keep in mind that in Tablet lingo "assembly" is referring to what we've been calling "mapped" or "bam". You'll also want to open the "Reference"

```
studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa .
```

Finally, once you've imported the "assembly" you'll want to click on "Import features" and open

```
studies/AthalianaReferences/resources/Athaliana_167_TAIR10.gene_exons.gtf .
```

Now take some time to look at the data. Check out your favorite gene perhaps. Can you see why long reads are so much better for cases of complicated alternative splicing? Do you see other differences between the data (besides that they are from totally different samples with different genes expressed)? Is the typical coverage distribution the same?

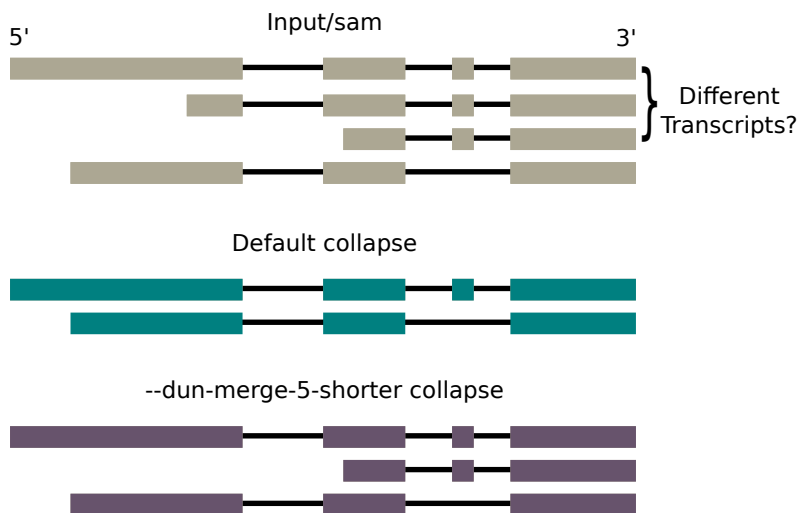
Once you've got a feel for the differences between Illumina and Iso-Seq data we'll move on to collapsing mapped reads into transcripts.

```
mkdir collapsed
# the next script requires a _sorted_ sam file, so one more conversion
samtools view runs/iseq/mapped/m16.flnc.sorted.bam > \
runs/iseq/mapped/m16.flnc.sorted.sam
# and the actual command to collapse
collapse_isoforms_by_sam.py --input runs/iseq/unpolished/m16.unpolished.flnc.fa \
-s runs/iseq/mapped/m16.flnc.sorted.sam -o collapsed/m16.flnc.to_genome \
--flnc_coverage 2

ls collapsed
```

To understand everything going on and the parameters for the previous step, we need to think a little bit about some of the wet lab details of Iso-Seq. In particular, reading a PacBio pamphlet, or just looking at the naming of "full length, non chimeric" reads, it seems like one could simply collapse every read that maps to the same position, perhaps allowing a small margin of flexibility for what counts as "the same". Certainly, this is part of what is going on, and the `--max_3_diff` of 100 bps is basically this "small margin of flexibility". But what about the `--max_5_diff` which defaults to 1000 bps? This must be, and is, much more lenient because of wet lab constraints. Nothing in the general Iso-Seq protocol actually has a way to tell if reads run from the 5' cap to the 3' poly-A tail, but rather, what PacBio defines as "full length" are reads that have the poly-A tail on one end, and the 5' adapter on the other end. The 5' adapters are attached with a blunt end ligation, so this does mean that reverse transcription successfully proceeded to the end of the piece of RNA. It doesn't mean that this was still an intact mRNA, and indeed, normally degradation will result in reads, particularly, from longer transcripts, that are slightly to severely truncated on the 5' end. That's why the collapsible margin is wider on the five prime end.

Some care should be taken here depending upon whether one is working with FLNC reads or high quality transcripts. This script appears to have been designed primarily for performing the second round of collapsing for the high quality transcripts. Therefore it's likely that both 5' and 3' margins should be increased when working with FLNC reads, although *most* of the time it seemed ok for the FLNC reads here. If you have high quality transcripts from `isoseq3 polish` that you trust a little more you could also set `--dun-merge-5-shorter`, which will prevent alignments from collapsing that differ on the 5' end by less than the margin, but which have a different set of exons.



Finally, the `--flnc_coverage 2` is *only* appropriate for FLNC reads, as it will cause every possible transcript that is supported by less than 2 alignments to be sorted into the 'bad' gff file while the rest end up in the 'good' gff file. With high quality transcripts, if the first step worked right for a transcript, you only would have a coverage of 1, but definitely want to keep it (and consider it 'good') anyways. We were skeptical this made sense for FLNC reads either, as particularly with a dataset like this that is a little low on coverage in general, there will be a lot of real transcripts with just one read. I personally hate throwing out precious data. However, the `collapse_isoforms_by_sam.py` tool does not have a complicated error model or anything behind it, it basically just applies a series of thresholds to see if some alignments should be collapsed. This means it treats a single unique transcript, which provides the only data for a genetic region the same as it treats a single transcript that is 'unique' only because it was erroneous and slightly misaligned at a splice junction. In the first case, we want to keep the single read, because it's all we have. However, in the second case, we definitely want to ignore the single outlying read and trust the majority at this locus. Luckily splitting the transcripts into 'good' and 'bad' allows us to handle them in the future with different trustworthiness and basically set the 'bad' set to be ignored at every locus where there was a 'good' model. This workaround is a bit unwieldy, and certainly not ideal, but it introduces fewer errors than without using `--flnc_coverage 2`, while keeping more data than just taking high quality transcripts.

Now you could stop here and just use the 'good' transcripts, or add one more step to merge in the non-conflicting set of 'bad' transcripts. That would provide a model for the transcribed parts of the genome. The recommend Iso-Seq tutorials kinda peter out here, and maybe there are projects where one would stop here, e.g. if you were running Iso-Seq to learn about alternative splicing in your favorite tissue, but in an otherwise well annotated species. That said, one of the most promising applications of Iso-Seq is to help define gene models for newly sequenced species. And here, where the goal is to describe full gene models, we really ought to take full advantage of the genome to help clean up, in particular, omissions in our transcript models.

For instance, the genome can help further when:

- None of the reads / transcripts for a locus were truly full length
- Transcripts for a locus were not expressed (enough to be measured) in any sequenced condition

OK, typically to define genes for a new genome one would bring in *all* the extrinsic data that is available, whether it was Illumina RNAseq, Iso-Seq, protein alignments from related species, ESTs, or something else and feed all of this to help support a *de novo* gene caller such as SNAP or Augustus that has a model for typical codon usage of genic and non-genic regions and can look for likely ORFs. This is somewhat organized and automated in public tools such as MAKER or PASA.

That said, that really starts to get beyond the scope of this course, so we will simply look at how you could use *just* the Iso-Seq data to support the *de novo* gene caller Augustus.

Hopefully, Iso-Seq will be incorporated more officially into gene calling pipelines in the near future.

Augustus can incorporate extrinsic evidence, like our Iso-Seq data, by adjusting its posterior probability of a particular call based upon the extrinsic data provided as 'hints'. That is to say, it runs its *de novo* prediction and then adjusts them as more or less likely based on the hints.

So we have to convert our collapsed gff file into a slightly different gff file that Augustus can understand as hints. Basically what we're doing here is converting the transcripts into a format where we can give AUGUSTUS more careful instructions on how to consider each part. Are the splice junctions as trustworthy as the aligned regions themselves? What about the transcription start site? Should exons and transcripts be considered only as a whole, or also as a part?

Let's just try it and look at the result.

```
# convert to gff3 (bc the hints script takes only gff3 correctly)
pfx=collapsed/m16.flnc.to_genome.collapsed.
gffread ${pfx}bad.gff -o ${pfx}bad.gff3
gffread ${pfx}good.gff -o ${pfx}good.gff3
# feel free to look at output with 'less' to see differences between gff and gff3

# convert our exons and transcripts to 'hints' for augustus
# note the 'good' hints (w/ 2+ reads) have a higher priority than the 'bad'
gff3_to_hints_isoseq.py -i ${pfx}good.gff3 -o ${pfx}good.hints.gff3 --priority=2
gff3_to_hints_isoseq.py -i ${pfx}bad.gff3 -o ${pfx}bad.hints.gff3 --priority=1

# finally we'll need the hints to be in one file
cat ${pfx}good.hints.gff3 ${pfx}bad.hints.gff3 > ${pfx}hints.gff3
```

Let's look at the 'features' column of the gff, and how they've now changed from the typical 'transcript' 'exon' that we had before.

```
# open file | get 3rd column | sort | count unique lines
less ${pfx}hints.gff3 | cut -f3 | sort | uniq -c
```

OK, so first, all 'ep' is very similar to the number of exons (if you run a similar count on the input files) and the features tts and tss are very close to the number of transcripts. The remaining features are in between. Huh, but the names are pretty cryptic, so what's what?

ass	Acceptor Splice Site, on 3' end of the intron (most commonly AG). Not our name for it.
dss	Donor Splice Site, on the 5' end of the intron (most commonly GT)
ep	Exon Part, so an exon with the edges trimmed (by <code>--trim_exonparts</code>) and Augustus will try to contain it in an exon, not match exactly
ip	Intron Part, so an intron with the edges trimmed (by <code>--trim_intronparts</code>) and Augustus will try to contain it in an intron, not match exactly
tss	Transcription Start Site
tts	Transcription Termination Site

I'd like to say we could run augustus now, but it would run overnight for the whole genome. So we're just going to run it for a subset of the genome.

The following script will cut a little chunk out of fa, gff3, and bam files alike. This is appropriate for the workshop or other niche cases like iterating through different parameters, etc to see what works best. But for the record, if you're doing this for the main run on a real project, please don't cut it smaller than your chromosomes/scaffolds.

```
# feel free to just type in place
official_gtf=studies/AthalianaReferences/resources/Athaliana_167_TAIR10.gene_exons.gtf

subset_genome_related.py --fasta studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa \
  -sChr1 -f1 -t3000000 --gff ${pfx}good.gff3,${pfx}bad.gff3,${pfx}hints.gff3,$official_gtf \
  --bam runs/iseq/mapped/m16.flnc.sorted.bam

# the script reports its output, but if it's easier
find -name *__Chr1*
```

Now we can run augustus

```
# I don't like typing the same things over and over again
where=Chr1_1-300000
mkdir gene_models
augustus --hintsfile=${pfx}hints__${where}.gff3 --species=arabidopsis \
  --alternatives-from-evidence=true --extrinsicCfgFile=extrinsic.E.cfg \
  --UTR=on --allow_hinted_splicesites=atac \
  studies/AthalianaReferences/resources/Athaliana_167_TAIR9__${where}.fa \
  > gene_models/flnc.${where}.augustus

less ${pfx}augustus
# the augustus output has the hints, commented protein sequence, explanation,
# and, what we are after, gtf lines with 'AUGUSTUS', the gene models.
# let's subset it to have just these.
less gene_models/flnc.${where}.augustus | grep AUGUSTUS > \
  gene_models/flnc.${where}.augustus.gtf
```

OK cool, we have our gene models, we recommend loading them into tablet (with the subsetted .bam and .fa files) and comparing them to the subsetted versions of the raw transcript (good/bad.gff3), and the official Arabidopsis annotation (studies/AthalianaReferences/resources/Athaliana_167_TAIR10.gene_exons.gtf).

Tablet is a little frustrating in that loading multiple 'transcript' features from different .gff's will pile them on top of each other in one line. You may a) open multiple tablet instances (think teamwork) or b) adjust the 'feature' column of the gff to have a different name. See below for an example

```
less gene_models/flnc.${where}.augustus.gtf | awk 'BEGIN {OFS = FS = "\t"} \
  { sub(/^/, "flnc.", $3) }1' > gene_models/flnc.${where}.augustus.gtf.tablet
less gene_models/flnc.${where}.augustus.gtf.tablet # check results
# don't worry about understanding all of the 'awk' command. The first bit
# is to tell it to use only tabs as a column separator. The second bit says
# find and replace the start of column 3 with "flnc."
# if you want to run this for the other files. The only part of the awk command
# you'll want to change is the "flnc." (e.g. to "original.")
```

Final note on viewing multiple gffs worth of feature sets in Tablet, you'll have to go to the features tab (blue triangle), click on 'select tracks', and then check all the tracks you want to see

In any case, congratulations for getting all the way to our first full gene models!

Mapping high quality isoforms We're now going to run (nearly) all the same steps for mapping high quality isoforms that we ran for mapping flnc reads above. As it is essentially the same pipeline, we are not going to give you the commands, although we will point out every instance where the commands should differ from above beyond the dataset being used. It is entirely up to you if, when and how much you check the intermediate output with e.g. `less` and `tablet`. For clarity, wherever you're supposed to fill in something it will be marked with `...` for a whole line or `< >` for a partial line.

Since you no longer have an exact script, we highly recommend filling in one as you go (e.g. in a text file)

```
# first, we need to look at the output from polishing
# if polishing did not finish, please stop it and copy the backup polishing results
ls polished
# The polished high quality draft transcripts are m16.polished.hq.fastq.gz
# extract them with 'gunzip'
...
# we'll start with gmap.
# we already have a database, so we don't have to run gmap_build, just gmap
# your input file will be 'polished/m16.polished.hq.fastq'
# please name the output in a way you know what it is and that it doesn't
# overwrite the previous analyses.
...
# we'll now need to sort, and convert back to sam
# think 'samtools view' for conversions, and 'samtools sort' to sort
...
# while you're at it and for later, please index your sorted bam
samtools index <your sorted bam>
# for the collapse_isoforms_by_sam.py, there are quite a few changes
collapse_isoforms_by_sam.py --input <the hq fastq file> -s \
    <the gmap sam output> -o <your output prefix> --fq
# we don't want a coverage filter as we had for the flnc reads,
# and we have to explicitly tell the script to parse fastq input (--fq)

# if you check your output, you'll see fewer output files than before,
# as without the --flnc_coverage parameter, the output won't be split into
# 'good' and 'bad'. Can you find the main output? It should be named
# <your output prefix>.collapsed.gff
# please convert this to gff3 with 'gffread'
...
# please convert the .gff3 file to hints using gff3_to_hints_isoseq.py
# note that you do not need to set --priority (but it also won't hurt)
...
# please subset the new bam and gff files with ./subset_genome_related.py
subset_genome_related.py --bam <your sorted bam> \
    --gff <your collapsed.gff>,<your hints.gff> -s Chr1 -f 1 -t 3000000
# and run augustus.
# All you will need to change are the hints and output files.
...
# and subset to just lines containing 'AUGUSTUS' with 'grep'
...
```

Alright, well done. You can get surprisingly far in executing a bioinformatics pipeline by taking an example from somewhere else that does approximately what you want, checking help functions to see if all parameters make sense for your data, and then plugging in your data (as you basically just did above). That said it is extremely important that you check as you go to see if the output is giving you what you expect, to read what information is available on the tools and generally try not to blindly trust it fits your use-case.

We'll do a larger scale comparison of the methods later, but feel free to compare the final gene models in tablet.

8.6.2 *de Novo*

So you've now seen two variations on reference based gene calling, and you've probably also gotten a chance to appreciate some of the major advantages, such as using the genome to fill in missing transcripts or 5' degraded transcripts.

But sometimes the genome is not available, or it can be so fragmented that many to most genes do not occur on one scaffold, or otherwise have quality issues. In such cases it can be preferable to use a *de Novo* approach.

That said, the only *de Novo* approaches we found very much consist of **under-development software** that hasn't yet been polished for an end user. In as much, we'll be approaching the next section as an **example of how you have to sanity check work as you go and be ready to fill gaps**.

The first step is, as with a genome based approach, to collapse draft transcripts to somewhat better draft transcripts. When a genome was available this made perfect sense, the genome brought in extra information to differentiate say alleles from paralogs. However, it is still quite necessary for a *de Novo* approach, as the first clustering step was intentionally conservative on the logic it is easier to collapse later than to deal with transcripts that have been too aggressively merged. Thus, the next clustering step with cogent is required to shift from 'under clustering' to a somewhat more aggressive 'best guess' sort of clustering.

The main commands we've taken from <https://github.com/Magdoll/Cogent/wiki/Running-Cogent>

Instead of having a *clustering* step, as under-development software, Cogent has a multi step process, with distance calculation, partitioning, and consensus calling all performed separately

```
# distance calculation:
run_mash.py -h
# looks like we really just need an input fasta file
# which we just need to extract from the polishing step
gunzip runs/isoseq/polished/m16.polished.hq.fasta.gz

run_mash.py --cpus 4 runs/isoseq/polished/m16.polished.hq.fasta
# check output
less runs/isoseq/polished/m16.polished.hq.fasta.s1000k30.dist
# you see pairs of transcripts with some info on
# distance and alignment length in the later columns.
# Seems reasonable. We'll also check the size of the file
# we don't know how large it is supposed to be, but I think
# we can say that if has fewer lines than input sequences or more
# than input sequences squared, we should worry.
wc --lines runs/isoseq/polished/m16.polished.hq.fasta.s1000k30.dist
```

In the next step we move from pairwise distances to partitions / clusters.

```
# partitioning
process_kmer_to_graph.py -h
# we'll need the input and output from above and we'll need
# to set an output directory & prefix
mkdir cogent
# also the wiki recommended using -c COUNTS_FILE, and showed an example
# COUNTS_FILE with (hq transcript ID, N flnc reads).
# Hmmmm, do we have the number for how many full length reads
# went into each high quality transcript?
less runs/isoseq/polished/m16.polished.hq.fasta
# we do, we just have to parse it out of the fasta headers
```



```

less runs/isoseq/polished/m16.polished.hq.fasta | grep '>' | cut -f1 -d';' | \
  sed -e 's/>///g' -e 's/full_length_coverage=//g' -e 's/ /\t/g' \
  > runs/isoseq/polished/m16.polished.hq.weights
# for help understanding the above command, please truncate it
# to the grep, check the output, and then start extending it piece by
# piece (grep, grep | cut, grep | cut | sed -e, grep | cut | sed -e -e, etc...)

# back to partitioning
process_kmer_to_graph.py -c runs/isoseq/polished/m16.polished.hq.weights \
  runs/isoseq/polished/m16.polished.hq.fasta \
  runs/isoseq/polished/m16.polished.hq.fasta.s1000k30.dist \
  cogent/ m16.polished.hq
ls cogent # puh, that's a lot of output
# let's keep looking at one folder
ls cogent/m16.polished.hq_0
less cogent/m16.polished.hq_0/in.fa
less cogent/m16.polished.hq_0/in.weights
# alright, it looks like we have a bunch of folders that contain the
# bits of fasta and weight assigned to their partition. Simple actually.

```

In the final Cogent step, the sequences within the partitions are finally assigned to transcripts and collapsed.

```

# consensus
reconstruct_contig.py -h
# from the wiki (more than the help function) we know
# this has to be ran for each partition. We'll use a for loop
for item in `ls cogent`;
do
  reconstruct_contig.py -p $item cogent/$item/;
done

ls cogent/m16.polished.hq_0/
# the final output is always in cogent2.renamed.fasta
# now lets sanity check the results\
# from the genome-based hq analysis we have some idea what to expect for
# number of transcripts
less collapsed/m16.hq.to_genome.collapsed.gff|awk '$3 == "transcript"|wc --lines
# so nearly 2k
# counting from this output format is a bit harder
# but ultimately is just concatenating all the fasta files, and counting the
# headers as we learned on the first day
cat cogent/m16.polished.hq_*/cogent2.renamed.fasta | grep '>' | wc --lines
# a lot less... hmmm...
# while it's same order of magnitude and _could_ be reasonable
# it was worrisome enough to start double checking
# let's count home many draft transcripts were still there after partitioning
cat cogent/m16.polished.hq_*/in.fa |grep '>'|wc
# and compare to total hq draft transcripts
less runs/isoseq/polished/m16.polished.hq.fasta | grep '>' |wc
# so barely over half of the sequences remained after partitioning.
# if we look at one of the meta info outputs from process_kmers_to_graph.py
less m16.polished.hq.partition.txt
# we see a) that the vast majority of partitions have 2+ sequences and
# b) a list of unassigned sequences, we can count these
less m16.polished.hq.partition.txt | grep unassigned | sed 's/,/ /g' | wc --words
# exactly the number we were missing. Success.

```

```
# OK, but what should we do with these, if we were making orthogroups,
# we would just skip the 'unassigned' singletons. But in this case, being a
# singleton doesn't even mean it's low quality or an outlier, just that the FLNCs from
# this draft transcript were successfully collapsed in the polish step, and cogent
# didn't have anything left to do.
# We already put together a convenience python script that will join
# collapsed and singleton transcripts together in a single file.
clean_cogent_output.py -f runs/isoseq/polished/m16.polished.hq.fasta -c cogent/ \
-o collapsed/m16.hq.de_novo.fa
# feel free to run with -h, or simply open if you want to know how it works
less collapsed/m16.hq.de_novo.fa | grep '>' | wc
# and now we have very comparable results to the genome based methods (numerically)
```

To be fair to the Cogent developers, this was all documented, and they do provide a way to get everything back together, see: [Using-Cogent-to-collapse-redundant-transcripts-in-absence-of-genome](#). That said, we missed the documentation in the first run, and noticing the discrepancy between expectation and result was therefore critical to getting the output right. Ultimately our solution was easier for the workshop, so we kept it.

OK, so we have transcript models now. Without a genome one certainly couldn't use e.g. augustus to extend or fill gaps in the sequences, what's missing is irrecoverable (without more sequencing). But we aren't done yet, the other thing that Augustus gave us was predictions for cds/protein sequences. For this we'll use ANGEL. When we were working with Augustus, we just used the pre-trained model for arabidopsis, ANGEL doesn't ship with pre-trained models, but it is easy to train.

First we need a training set, we'll just take a naive ORF caller (you know, based on start/stop codons and length), and grab the transcripts with the longest predicted CDS. Then we'll filter them, e.g. to avoid having two nearly-identical transcripts for training. Then we'll train.

```
mkdir angel

dumb_predict.py -h
# since this is such a small dataset we set --min_aa_length kinda low
dumb_predict.py collapsed/m16.hq.de_novo.fa angel/m16.hq.de_novo --min_aa_length=290
ls angel/ # we'll continue with *.final.*
# pick low-redundancy training set
angel_make_training_set.py -h
angel_make_training_set.py angel/m16.hq.de_novo.final angel/m16.hq.de_novo.final.train
ls angel/ # we'll continue with *.train.cds and *.train.utr
# and train, this will take a minute
angel_train.py -h
angel_train.py angel/m16.hq.de_novo.final.train.cds \
  angel/m16.hq.de_novo.final.train.utr angel/arabidopsis.pickle --cpus=4
# our trained model is now at angel/arabidopsis.pickle, importantly the computer
# can read it, even if we can't
```

Now that we have our trained model, we just need to run it on all the sequences

```
angel_predict.py -h
# angel can only output predictions to the same directory
cd angel/
# this will take several minutes
angel_predict.py ../collapsed/m16.hq.de_novo.fa arabidopsis.pickle m16.predictions \
  --output_mode=best --min_angel_aa_length 100 --min_dumb_aa_length 100
ls
cd ..
```

Now we have our completed gene models!

8.7 Comparing and Evaluating Gene Models

And with completed gene models, whether or not one has several methods to compare, one definitely wants some feedback on whether they are any good or not.

We'll run through a few examples of basic ways to evaluate the results. **We'll just provide the first command, and then let you figure out how to change it / if it makes sense for the rest of the methods.**

Exactly what you compare is up to you, you could compare the results of processing the high quality draft transcripts in a *de novo* or reference based manner. You could compare the augustus output when it gets hints from the flnc reads, the hq transcripts, or is run without hints. You could compare any of the methods to the official *Arabidopsis* gene models. You could compare the genome-based IsoSeq transcripts to the Trinity assembly (minidata.fastq from day 1 consisted of just reads that mapped to Chr1:1-300000). There are a lot of options.

As it can be hard to compare just the genome fragment we used for Augustus to the full genome/transcriptome predictions, we've provided full augustus results if you want them.

```
cp -r ../BackUpData/augustus ./
```

First we need to have actual transcript/protein sequences, these are easy to create from the gtf/gffs.

```
# get transcript sequences from gff with just transcripts/exons
# e.g. our collapsed gff sequences
gffread collapsed/m16.flnc.to_genome.collapsed.good.gff -O \
  -g studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa \
  -w collapsed/m16.flnc.to_genome.collapsed.good.transcript.fa

# get transcript, cds, and protein sequences from a full gff (e.g. from Augustus)
basename=studies/AthalianaReferences/resources/Athaliana_167_TAIR10.gene_exons
gffread ${basename}.gtf -g studies/AthalianaReferences/resources/Athaliana_167_TAIR9.fa \
  -w ${basename}.transcript.fa -x ${basename}.cds.fa -y ${basename}.protein.fa
```

Next we'll want to run a numerical comparison. How many transcripts/proteins were produced and how long were they?

```
# quast is a program that can perform a fairly thorough quality control
# on a variety of DNA sequences / assemblies.
quast.py angel/m16.predictions.ANGEL.cds --min-contig=1 -o quast/angel_cds
# but quast will only work for DNA sequence, for proteins we can use the
# count_fasta.pl from the first day. Copy it into the directory, then
./count_fasta.pl -i 20 angel/m16.predictions.ANGEL.pep
```

Knowing you have long sequences is nice, and it's particularly encouraging when you have longer protein sequences. However, sometimes and particularly when there is no reference available, it can be hard to interpret. Moreover, given an 'omics sized analysis, you'll always have the occasional long ORF by chance, etc. So once one has a short list of methods that produced decent *quantitative* results, it's good to perform a more *qualitative* check. This can include steps you've seen before, like *looking* at the models with Tablet. But another important question is always whether the sequences can be annotated.

In plant biology, Mercator (and the newly released Mercator 4) provide a nice quick way to get functional annotations. You can upload your sequences to <http://www.plabipd.de/portal/web/guest/mercator4>, and the results should be there within a few minutes. That said, if everyone does this at once, it may take a good deal longer, so feel free to look over each others shoulders.

The web interface provides some summary output, but most importantly you can download the results as a tab separated annotation file and then calculate whatever you need.

```
# count all transcripts
less <your results> |grep "T$" |wc --lines
```

```
# count annotated transcripts
less <your results> |grep "T$" |grep -v "not assigned" |wc --lines
```

One could similarly look at assigning pfam domains, assigning GO terms, or comparing transcripts directly to other related species whether via reciprocal best blast or orthogroup creation.

Challenge Assignment. If you've made it this far, and we haven't yet interrupted you to start the Q&A session, you might want to take this time to try and apply some of your R skills to the comparison above. Can you make a plot of the N50 or total length for your comparisons? Can you make a comparative plot that summarizes the Mercator/MapMan annotations?

8.8 Future perspectives

We spent a lot of this section talking about how this was un-stable software, how it was changing a lot, how to check as you go when working with new pipelines, etc... You might almost get the idea that we don't expect you to be able to use this script as an appropriate guideline 6 months from now, and you'd be right.

Here are some things that we think are needed, and we think will come *soonish*. These are things I would definitely look for at the start of my next project.

- Generally, there will be a flux, better tools will come out, tools will change names and parameters
- Some equivalents of FastQC that give more tailored feedback for PacBio / Nanopore data will become available
- Cogent or a similar tool will likely provide a 1-step process for genome-free draft transcript collapsing
- More tools will become available that are specialized for direct RNAseq
 - Genome-based collapse for Nanopore reads that has poly-A tail detection on the 3' end and, of course, accounts for the higher / biased error.
 - Read and or transcript polishing (self but very importantly also Illumina based) that are explicitly aware of the dynamic range of transcriptomics data
- There's some interest in 5' and 3' selection in the Iso-Seq pipeline, which already even has some early software available for the collapse step [TAMA](#)
- Finally, in good time, both Iso-Seq and direct RNAseq will be incorporated into full gene annotation pipelines.

9 Questions and answers

Congratulations, you've made it through the workshop material!

Do you have questions on what we've covered?

Do you have questions on transferring the information here to your own data?