Organizing Data Science Projects

NIH BYOB, 2019-08-22

Justin M Fear

- IRTA Fellow (NIDDK/NIH)
  - Genomics
  - Gene Regulation
  - Drosophila
- Contact:
  - @jfear
  - justin.fear@nih.gov
  - Justin Fear@nih-byob.slack.com
- Slides are at:
  - http://talks.geneticsunderground.com

Why Project Organization

Find things quickly

(≤ 10 min)

- Find the exact code used to generate result
- Tweak a plot
- Jump back into a project picking up where you left off

Reuse code

Automate the boring stuff

- Standardized workflows
- Modularize code for testing, maintenance, and usability
- Save brain power for fun tasks

Share code and results

- Quickly show a collaborator what you did
- Summarize side analyses for archive

Recover from data disasters

- Oops we swapped sample names now you need to re-run
- We forgot to tell you about these addition 10 samples
- The file we sent you was truncated

Reproducible Research

Have you seen this?

Please don’t do this

Poor uses of file names

```
.
├── deg_jmf_final_v2.sh
├── deg_lmm_final.sh
├── deg_lmm_v1.sh
├── deg_lmm_v2.sh
├── deg_step1_jmf_v1.sh
├── deg_step2a_jmf_v1.sh
├── deg_step2_jmf_v1.sh
├── deg_step3_jmf_v1.sh
├── deg_step3_jmf_v2.sh
├── deg_step3_lmm_v1.sh
├── download_data_v1.sh
└── download_data_v2.sh
```

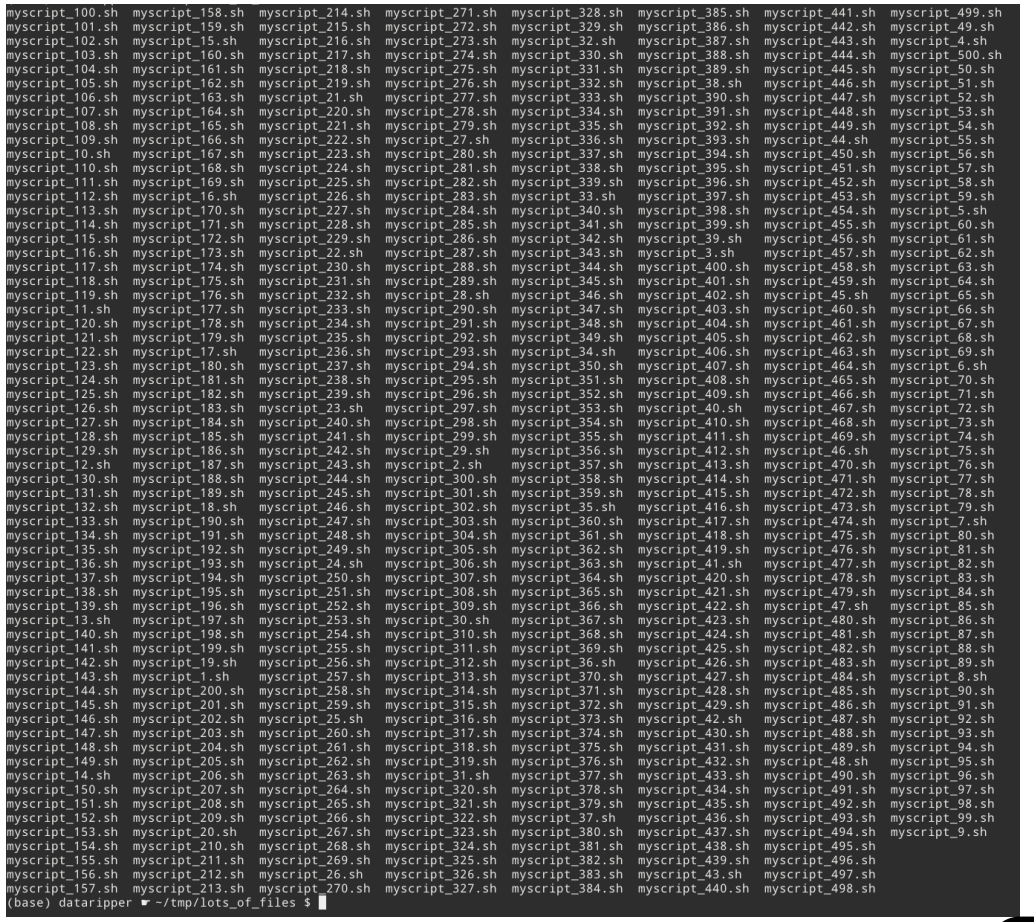
- File names are not great for version control

> If you name something final, you will always have another version.

- File names are not great at ordering steps in a workflow

> Adding or re-ordering steps is confusing at best.

Make file names descriptive and concise.



Poor uses of folders

- One folder to rule them all

> For small projects you can get by with putting everything in a single folder. But, once you have more than 20 files it is hard to find things. Using search is only useful if you know what you are searching for.

- Too many folders

> As above, once you get more than 20 or so folders is becomes hard to find things.

Make your own folder hierarchy and stick to it.

Poor uses of scripts

```
# Run this part first
# bunch of lines of code

# Run this part second
# bunch of lines of code

# Run this part third
bunch of lines of code

# Run this part fourth
# bunch of lines of code
```

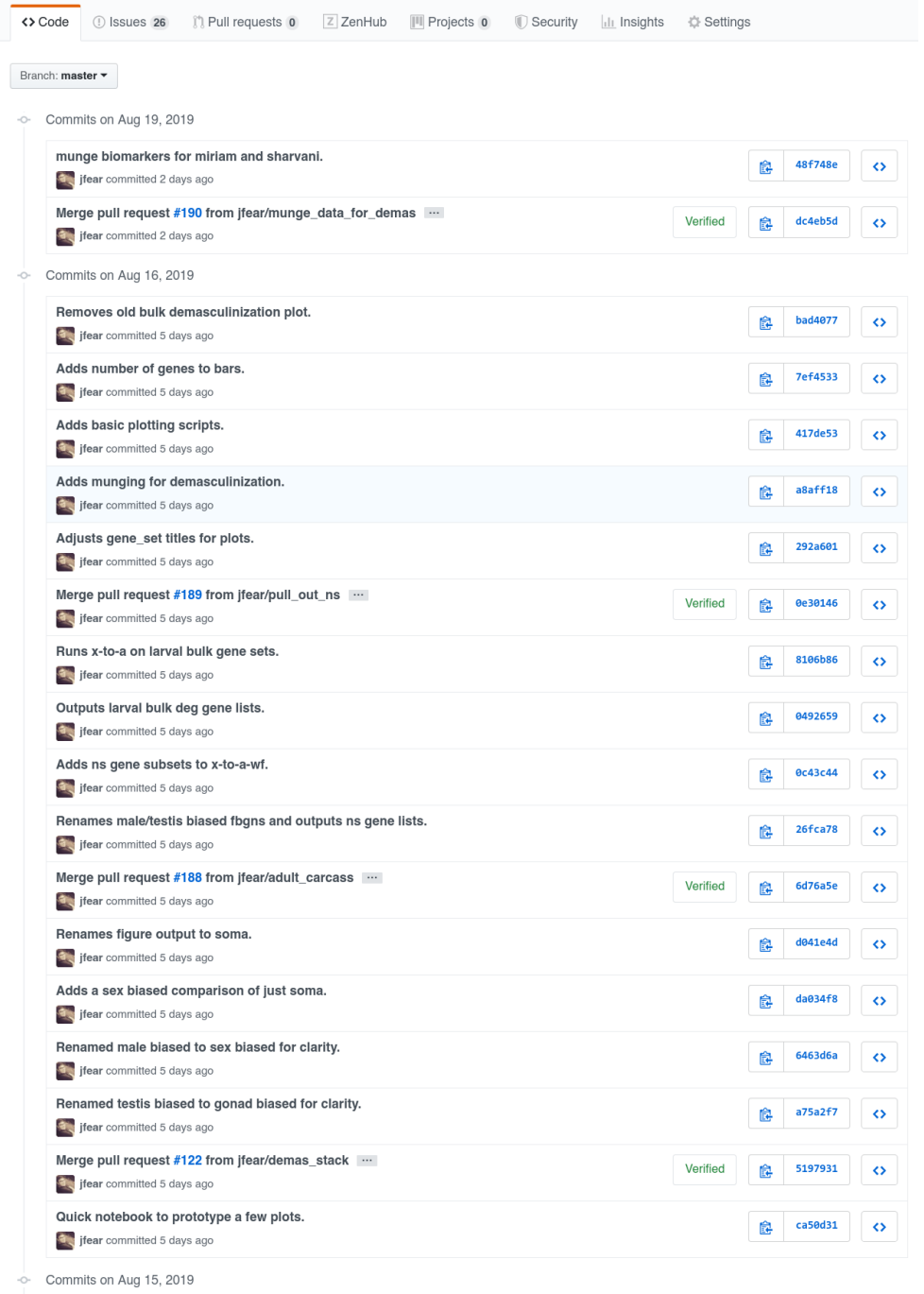
- Don't comment and uncomment parts of script

> Your-future-self will have no clue what was actually run

- Don't copy and paste from a script

> Beginners often has scripts with lots of comments describing each step. They the copy and paste from the script onto the command line. Like commenting and uncommenting you may not know exactly what you did.

How to get organized



Tools of the trade (VCS)

Version Control Software

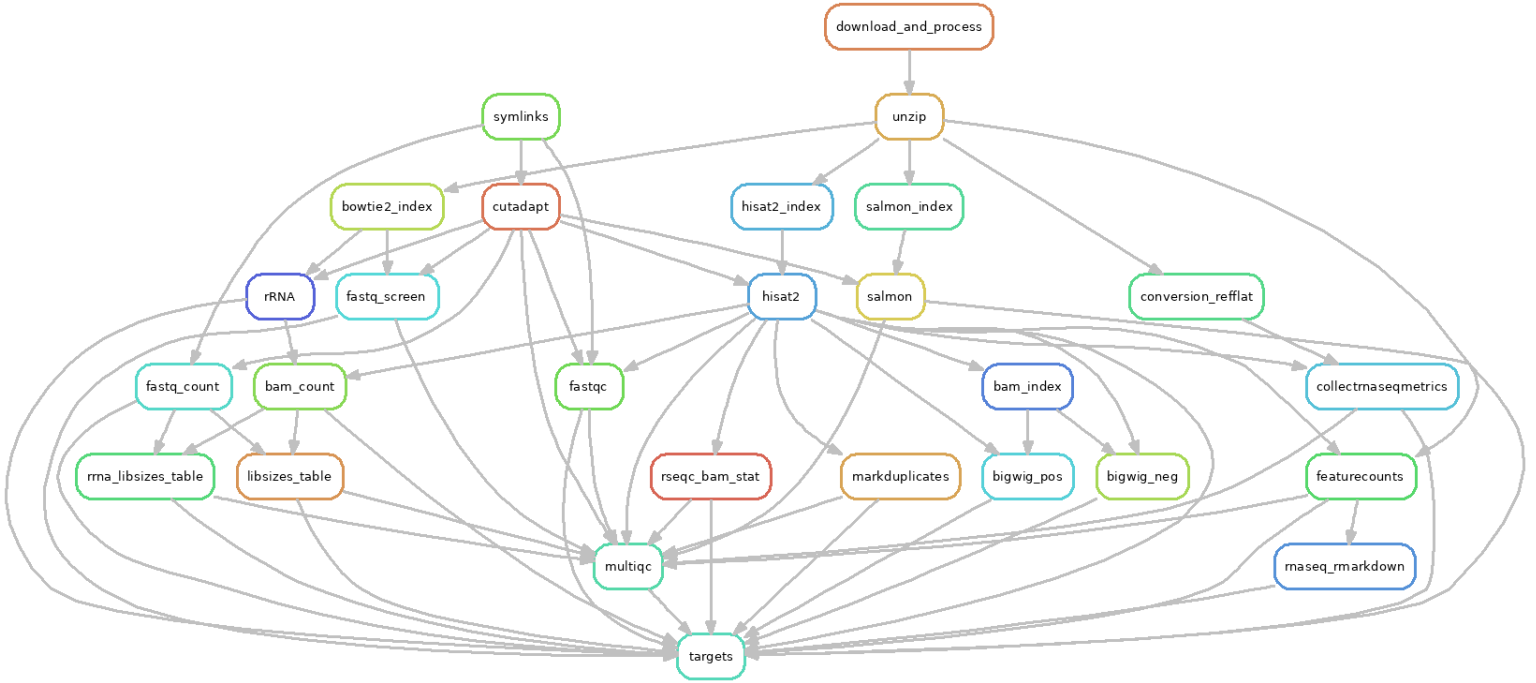
- Use VCS such as git

> VCS is like MS Words track changes

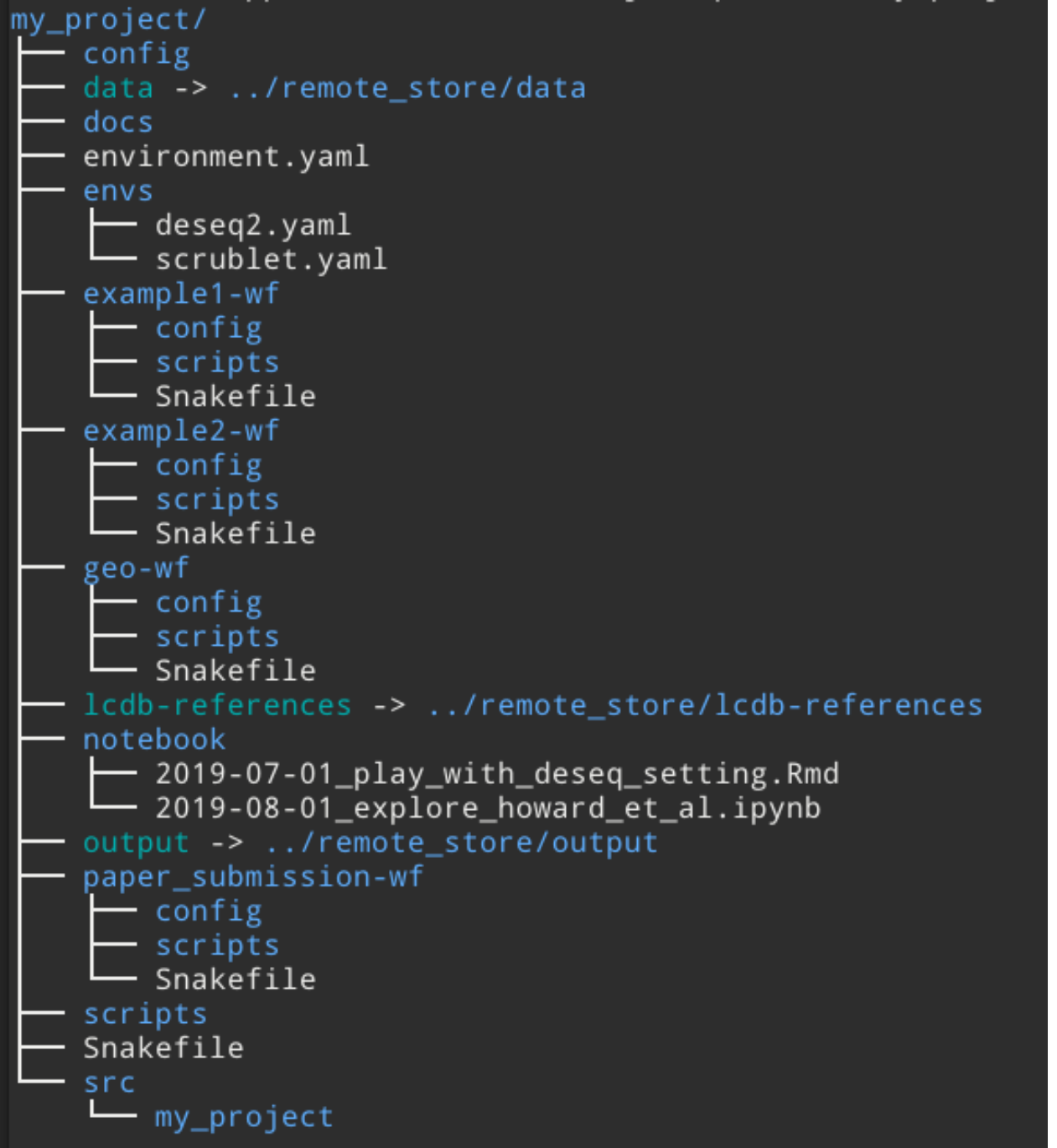
- Use the cloud to store to your code

> Github, GitLab, and Bitbucket are common places to store your version controlled code.
>
> You can make code private or public, and it is available anywhere with internet.

Tools of the trade (Workflows)



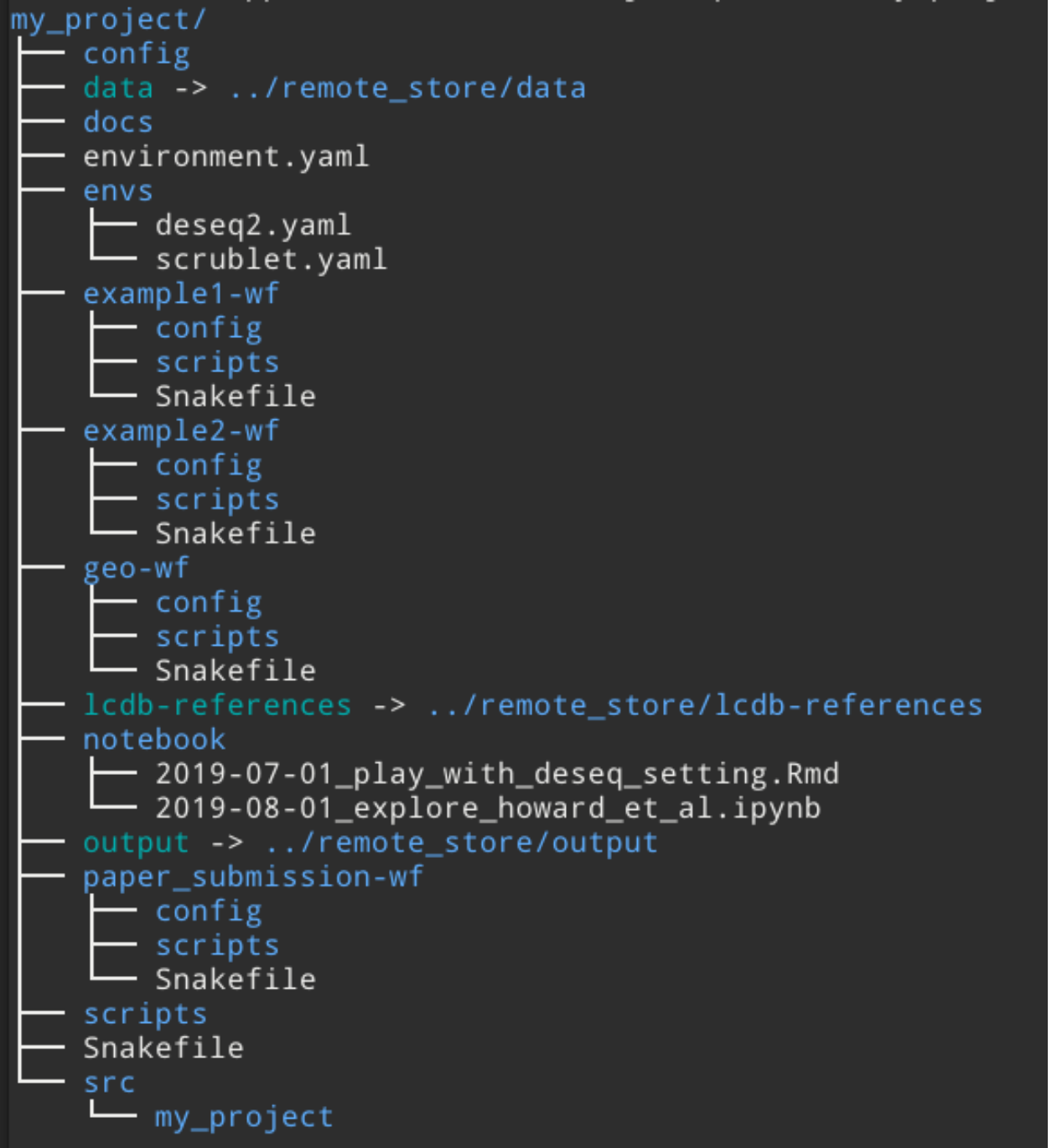
- Orchestrate analysis using workflow software
  - Galaxy
  - Make
  - Snakemake



Project Folder Organization

- Folder structure is personal preference
- Folder names are personal preference

There are general *best practices*



1. Use the same folder structure and names across projects

- Everything has its place
- You can quickly move between projects

2. Separate original data, generated data, and scripts

Not stored in version control

```
├── data # original and external
├── lcdb-references # multi-project
├── output # generated output
│   ├── example1-wf
│   ├── example2-wf
│   └── paper_submission-wf
```

- Improves mobility
- Delineates what you generated
- Allows reuse of common data across projects

```
data # original and external
├── external
│   ├── DroID_DPiM_2018-03-29.txt # website
│   ├── Ferrari_et_al_2006.tsv # paper
│   ├── Ferrari_et_al_2006.readme # paper details
│   ├── FlyBase/ # community
│   └── maria/ # collaborator
├── rnaseq_samples # our data
│   ├── ...
│   └── w1118_LG_m_r4_B_C12.fastq.gz
└── singleCellSeqData # out data
    ├── ...
    └── SV_9_10X_Te/
```

3. Uses workflows to orchestrate

```
./example1-wf
├── config
│   ├── config.yaml
│   └── sampletable.tsv
├── scripts/
└── Snakefile
```

```
from larval_gonad.config import read_config

common_config = read_config('../config/common.yaml')
sample_table = pd.read_csv('../expression-atlas-wf/config/sampletable.tsv', sep='\t')
sample_attrs = sample_table.samplename.str.extract('(?P<species>\w+)_(?P<tissue>\w+)_(?P<sex>\w+)_(?P<re
SPECIES = sample_attrs.species.unique().tolist()
TISSUE = sample_attrs.tissue.unique().tolist()


rule all:
    input:
        '../output/neox-wf/sturgill_2007.xls',
        '../output/neox-wf/sturgill_2007_mullerD.feather',
        '../output/neox-wf/sturgill_2007_mullerE.feather',
        expand(
            '../output/neox-wf/{species}_muller_{suffix}.feather',
            species=['dpse', 'dwil'],
            suffix=['A', 'D', 'E']
        )


rule download_strugill:
    output: '../output/neox-wf/sturgill_2007.xls'
    shell: """
        curl -o {output[0]} https://media.nature.com/original/nature-assets/nature/journal/v450/n7167/ex
    """


rule sturgill_data_prep:
    input: rules.download_strugill.output[0]
    output:
        mullerD='../output/neox-wf/sturgill_2007_mullerD.feather',
        mullerE='../output/neox-wf/sturgill_2007_mullerE.feather'
    script: 'scripts/parse_sturgill.py'


rule muller_arm_movement_classes:
    input: '../output/expression-atlas-wf/muller_arm_assignment.feather'
    output:
        muller_a = '../output/neox-wf/{species}_muller_A.feather',
        muller_d = '../output/neox-wf/{species}_muller_D.feather',
        muller_e = '../output/neox-wf/{species}_muller_E.feather'
    script: 'scripts/muller_arms_classes.py'
```

4. Modularize reusable code

```
lcdb-wf@56c948d  #submodules
src/    # project level package
├── my_project
│   ├── io.py
│   ├── plotting.py
│   └── stats.py
├── tests/
│   ├── test_io.py
│   └── test_stats.py
└── setup.py
```

```python
def decompress_seq(x: int, length=16):
    """ Un-pack a DNA sequence from a 2-bit format

    Based on code from: https://github.com/10XGenomics/cellranger

    cellranger/lib/python/cellranger/utils.py

    Parameters
    ----------
    x : int
        Number sequence to be decoded.
    length : int
        Length of the barcode. This can be found in the molecular info hdf5
        file from 10x genome.
        molInfo.get_node_attr('/metrics', 'chemistry_barcode_read_length')

    """
    bits = 64
    x = np.uint64(x)
    assert length <= (bits / 2 - 1)
    if x & (1 << (bits - 1)):
        return "N" * length
    result = bytearray(length)
    for i in range(length):
        result[(length - 1) - i] = bytearray(NUCS[x & np.uint64(0b11)].encode())[0]
        x = x >> np.uint64(2)
    return result.decode()


def two_bit_mapper(iterable):
    """Return a dictionary mapping 2bit encoded Seqs.

    Parameters
    ----------
    iterable : list-like
        Unique list of 2bit encoded sequences.

    Returns
    -------
    dict : Mapper from encoded to decoded

    """
```

5. Use a style guide and linters

- Consistent style improves readability
- Just google my language and style guide

- Linters catch syntax erros and point out style problems.

  - `flake8 # python`
  - `lintr # R`

- Fix ugly code with software

  - `black # python`
  - `styler # R`

```
for (i in seq(10)) {
for (j in seq(100)) {
if (i == j) {
print(TRUE)
} else if (i %% j == 0) {
print("modulo")
} else {
print(FALSE)
}
}
}
```

```
for (i in seq(10)) {
  for (j in seq(100)) {
    if (i == j) {
      print(TRUE)
    } else if (i %% j == 0) {
      print("modulo")
    } else {
      print(FALSE)
    }
  }
}
```

6. Split out configuration for consistency

```
./config   # Project config
    ├── common.yaml
    ├── gene_sets.yaml
    └── colors.yaml

./example1-wf   # Workflow config
    ├── config
    │   ├── config.yaml
    │   └── sampletable.tsv
```

Project config

Contains info that is needed across the project.

- Project name and github url
- Assembly and Annotation
- alpha level

Workflow config

Anything you may tweak in the future.

- Various thresholds
- Workflow specific references
- Various Mappings (i.e. file name to title)

7. Use containers and environments (portability and

One of the hardest problems in data science is managing software.

```
./environment.yaml  # project env

./envs   # specific tools conda envs
    ├── deseq2.yaml
    ├── scrublet.yaml
    ├── seurat2.yaml
    └── seurat3.yaml
```

Containers (Docker, Singularity)

- Completely reproducible system
  - Kernel and Software

Environments (Conda, pipenv)

- Install and manage software versions
- Different versions of software can be installed in different environments

8., 9., 10. Documentation

What is not documented, stays not documented

What to document (Everything!)

- How was the data generated
- Record all “experiments”
  - failed attempts
  - comparing different methods
- Record the reasoning for any decision points
- Clearly describe how to get final results

Where to document (Everywhere!)

- Sample/Resource Table
- README
- Top of scripts
- Function/Class Docstrings
- Code comments (but not too many)
- Literate Programming (i.e. notebooks)
- Project Blog

Sample Table

```
./example1-wf
    ├── config
    │   ├── config.yaml
    │   └── sampletable.tsv
```

A1_OCP	....A1_OCP_1.fastq.gz	OCP	A1	A	1	25	f	0	1	0	A
A6_TCP	....A6_TCP_2.fastq.gz	TCP	A6	A	6	15	m	1	0	0	B

Add as much information about your samples.

Top of Scripts

- Describe what the script does
- Any major decisions that you made
- Anything to help you remember

```
"""Decide what cells to use.      Justin Fear, 2 months ago • Refactored cellselection.

Before doing downstream analysis, I need to decide what cells to use. There are four different
steps of cell selection.

1. Remove empty cells
2. Remove heterotypic doublets
3. Remove homotypic doublets
4. Remove low complexity cells (or high mitochondrial expression, or high rRNA expression)

STEP 1: There are several tools available for identifying empty cells. All of these tools model
total UMI and find a threshold to separate GEMs with cells and GEMs without cells. Here I am
three different tools (cell ranger v2, cell ranger v3, dropLetUtils).

DECISION: I will go forward using the consensus of cell ranger v3 and dropLetUtils.

STEP 2: There are several tools available to identify heterotypic doublets. All of these methods
generate synthetic doublets by mixing cells from different clusters. They then re-cluster and call
cells that cluster with the synthetic cells doublets. I have decided to only use scrublet because
it uses the raw count data directly while other method require a pre-clustered datasets.

DECISION: I remove cells scrublet calls as doublets

STEP 3 + 4: Homotypic doublets are impossible to identify bioinformatically. The only signal would be
cells with high UMI counts and a large number of expressed genes. Similarly, cells with low complexity
I need to remove cells with low gene search: str ntially cells with high mito or rRNA expression.
To look at this problem I use a grid search approach where I try the combination of different filtering
thresholds and compare clustering of these cells.

DECISION: Filtering of mitochondrial and rRNA had no affect on clustering so will be ignored. The low end
cutoff of 200 and 500 expressed genes had very similar results, while a more extreme 1,000 genes behaved very
differently. Because we have potentially quisent cell types I will use the 200 gene cutoff. The high cutoffs
of 5,000 and 6,000 were similar while no cutoff performed very differently suggesting these high end genes we
driving clustering. I will use 5,000 because these high expressing cells are potentially doublets.

"""
import os
import yaml
from itertools import chain
from pathlib import Path

import pandas as pd
from snakemake.shell import shell

from larval_gonad.config import read_config
from larval_gonad.io import pickle_dump


configfile: 'config/config.yaml'
common_config = read_config('../config/common.yaml')
ASSEMBLY = common_config['assembly']
TAG = common_config['tag']

SAMPLES = pd.read_csv('../config/scrnaseq-sampletable.tsv', sep='\t', index_col=0).index.tolist()
```

Functions and Classes

- Any function you will call from another script.
- Add type hints if it is confusing what goes in.
- Add examples to clearly show what the function does.

```python
def decompress_seq(x: int, length=16):
    """ Un-pack a DNA sequence from a 2-bit format

    Based on code from: https://github.com/10XGenomics/cellranger

    cellranger/lib/python/cellranger/utils.py

    Parameters
    ----------
    x : int
        Number sequence to be decoded.
    length : int
        Length of the barcode. This can be found in the molecular info hdf5
        file from 10x genome.
        molInfo.get_node_attr('/metrics', 'chemistry_barcode_read_length')

    """
    bits = 64
    x = np.uint64(x)
    assert length <= (bits / 2 - 1)
    if x & (1 << (bits - 1)):
        return "N" * length
    result = bytearray(length)
    for i in range(length):
        result[(length - 1) - i] = bytearray(NUCS[x & np.uint64(0b11)].encode())[0]
        x = x >> np.uint64(2)
    return result.decode()


def two_bit_mapper(iterable):
    """Return a dictionary mapping 2bit encoded Seqs.

    Parameters
    ----------
    iterable : list-like
        Unique list of 2bit encoded sequences.

    Returns
    -------
    dict : Mapper from encoded to decoded

    """
```

NUCS: *list*

Literate Programming

```
./notebook
    ├── 2019-08-01_bulk_deg.Rmd
    └── 2019-08-10_bulk_ma.ipynb

./docs
    ├── cell_number_counts.ipynb
    └── permutation_summary.ipynb
```

- Jupyter Notebooks
- R Notebooks and Rmarkdown

https://github.com/markusschanta/awesome-jupyter

In [2]:
```
# Imports
import numpy as np
import pandas as pd

from IPython.display import HTML, Image

import qgrid

import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

from lcdblib.images.SVG import nb_svg

# Module settings
pd.options.display.max_columns = 999
sns.set_context('poster')
```

I have have been exploring the TaDa data and playing with four different analysis methods.

- Clough, Emily, Erin Jimenez, Yoo-Ah Kim, Cale Whitworth, Megan C. Neville, Leonie U. Hempel, Hania J. Pavlou, et al. 2014. “Sex- and Tissue-Specific Functions of Drosophila Doublesex Transcription Factor Target Genes.” Developmental Cell 31 (6): 761–73. doi:10.1016/j.devcel.2014.11.021.
- Marshall, Owen J., and Andrea H. Brand. 2015. “Damidseq_pipeline: An Automated Pipeline for Processing DamID Sequencing Datasets.” Bioinformatics (Oxford, England) 31 (20): 3371–73. doi:10.1093/bioinformatics/btv386.
- Carl, Sarah H., and Steven Russell. 2015. “Common Binding by Redundant Group B Sox Proteins Is Evolutionarily Conserved in Drosophila.” BMC Genomics 16: 292. doi:10.1186/s12864-015-1495-3.
- Maksimov, Daniil A., Petr P. Laktionov, and Stepan N. Belyakin. 2016. “Data Analysis Algorithm for DamID-Seq Profiling of Chromatin Proteins in Drosophila Melanogaster.” Chromosome Research: An International Journal on the Molecular, Supramolecular and Evolutionary Aspects of Chromosome Biology, October. doi:10.1007/s10577-016-9538-4.

Each pipeline has its advantages and disadvantages. The `Marshall` method is not statistically rigorous and can only handle duplicates, but it has an easy to run pipeline. The `Clough` and `Carl` are more statistically satisfying, but their implementation requires deciphering the papers. The main difference between these two methods are how reads are counted (`Clough` uses 500bp windows while `Carl` extends reads and counts overlapping GATC sites).

Both `Carl` or `Clough` methods reasonable, but in the end I felt both counting methods are wrong (discussed later). I have devised my own counting method, which is similar to the new `Maksimov` method, but use the same analysis approach as `Carl` and `Clough`.

For me to effectively argue for my method over other published methods, it is important to understand these data. Next I visualize select steps of the DamID protocol (DamID in Living Color). Then I show some real data (Example Data) at the genome level. Finally I show some comparisons of the different methods and hopefully convince you that my proposed method seems the most reasonable.

DamID in Living Color

In [3]:
```
# Get dam ID SVG
s = nb_svg('../../figs/drawings/dam_cuts.slides.svg')
#s.getLayers()
```

Getting a gut feeling for what to expect from the data is important for data science. It was extremely valuable to participate with Miriam in DamID and library construction. I also needed to visualize the DamID process, which lead to some insight. Here I show some key steps in a DamID experiment.

Original Seqnece

Dedicated Project Blog

- Aggregate notebooks
  - `bookdown # R`
  - `jupyter webbook # python`
- Various static site generators
  - Pelican
  - Nikola
  - jekyll
  -

`cellranger3-wf` uses cell ranger v3 with default settings. Recently, 10X Genomics released version 3 of cell ranger. This version adds an additional step to the cell calling algorithm which improves calling of low RNA content cells. Interestingly, this method called more cells than `cellranger-force-wf` in all replicates expect **Testis 1**.

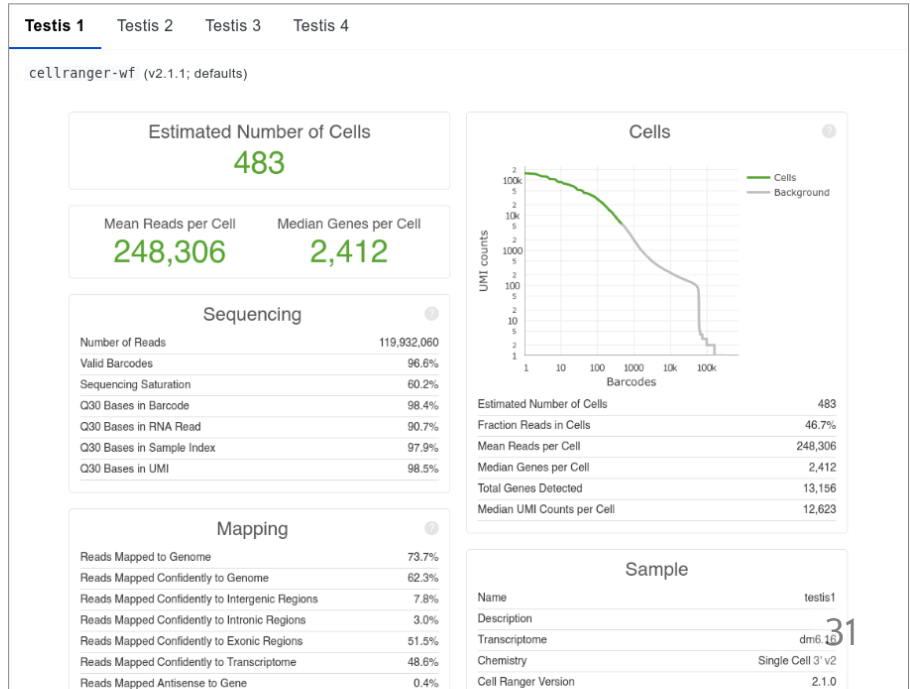
`droputils` uses DropLetUtils v3.0.1 with default settings. DropLetUtils is an R packaged with several algorithms for scRNA-Seq. I used the `emptydrops` function which models UMI to classify empty GEMs. This method performs very different for each replicate. This method seems to over call cells in **Testis 1**, **Testis 3**, and **Testis 4** while performing similarly to `cell ranger` on **Testis 2**.

Without a ground truth data set it is impossible to know which method is best approximating real cell calls. Consensus is often used in the absence of truth. However, sense all of these methods use total UMI in their models, the 4-way consensus would be the same as the most conservative cell calls (`cellranger-wf`). Given the known limitation of `cellranger-wf` and the *ad hoc* nature of `cellranger-force-wf` I decided to use the consensus of `cellranger3-wf` and `droputils` .

Table 2. Cell count after removing empty GEMs.

Replicate	`cellranger-wf`	`cellranger-force-wf`	`cellranger3-wf`	`droputils`	2-way Consensus‡
Testis 1	483	3,000	2,826	13,884	2,790
Testis 2	550	3,000	6,385	5,765	4,801
Testis 3	423	8,000	12,485	39,872	12,515
Testis 4	349	8,000	15,033	34,761	15,033

‡Intersection of `cellranger3-wf` and `droputils` .



10 Best Practices

1. Use the same structure and names across projects
2. Separate original data, generated data, and scripts
3. Use workflows to orchestrate
4. Split out configuration for consistency
5. Modularize reusable code
6. Use a style guide and linters
7. Use containers and environments
8. Document as you go
9. Document as you go

Resources

- Cookiecutter
- Github
- Bitbucket
- Snakemake
- Data Science git-flow