DRAPER®

Pipelining in Python

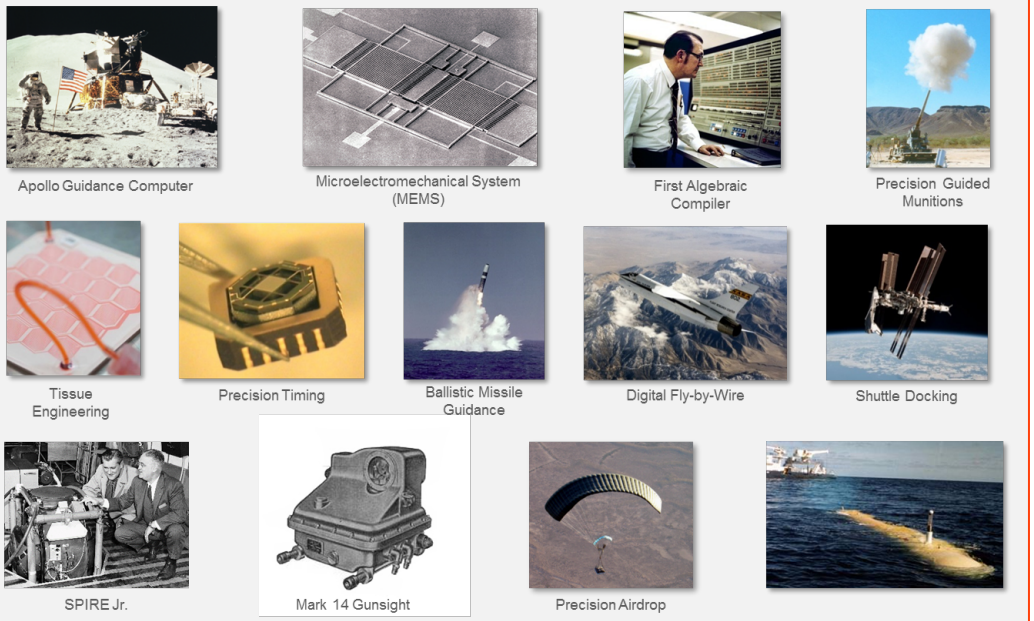
Using Snakemake with biological applications

Laura Seaman Ph.D.

April 15th, 2020

The Charles Stark Draper Laboratory, Inc.
555 Technology Square, Cambridge Mass. 02139-3563
CAGE Code: 51993

About Draper



Independent not-for-profit research and development company

- Headquartered in Cambridge, MA
- 1500+ employees, $520 million revenue
- Serving the nation since 1932
- Independent organization since 1973

Overview

- **Intro to Snakemake**
- Snakemake Syntax
- Snakemake examples
- Snakemake in Research: Detecting genetic engineering

Data processing pipelines

Transform data from one state to another

- Repeatable
- Scalable
- Versioned
- Monitored

Uses

- Convert data to a common format
- Move data between databases
- Prepare data for investigation and imaging

Specifically

- Process financial / stock market data
- Analysis of genetic data for identifying engineered organisms
- Analyze user activity on websites

Snakemake is a workflow management system

- Provides structure for data analysis pipelines
  - Process data files: sequencing data, twitter posts, financial reports, …
  - Goes through a set of consistent steps to produce an output
- Python based
- Encourages modularity, reproducibility, and scalability

Snakemake is good at:

- Incorporating multiple programs or environments
- Encourages modularity, reproducibility, and scalability

Does not:

- Include versioning or testing
- Work well with varied or inconsistent naming conventions
  - This is how snakemake figures out what to do

Overview

- Intro to Snakemake
- **Snakemake Syntax**
  - Parts of the code
  - Running the code
- Snakemake examples
- Snakemake in Research: Detecting genetic engineering

Rules define the flow

```
rule map_reads:
    input:
        R1="samples/{sample}_L001_R1_001.fastq.gz",
        R2="samples/{sample}_L001_R2_001.fastq.gz",
    output:
        sam="aligned_reads/{sample}_aligned.sam",
    shell:
        """
        bowtie2 -x {REF_GENOME} -p {core} -1 {input.R1} -2 {input.R2} -S {output.sam}
        """
```

Rules define the flow

Input
- Files (or directories) needed for a rule to start.
- If these do not exist in the current file structure or the output of another rule, it will fail to start.

Output
- The files created by the rule
- If execution ends and these files are not all created, the job fails
    - Any of these files that were created are deleted

Execution
- Shell: a series of bash commands
- Run: a series of python commands to execute
- Script: run a python or R script
- Notebook: passes snakemake object to a Jupiter notebook

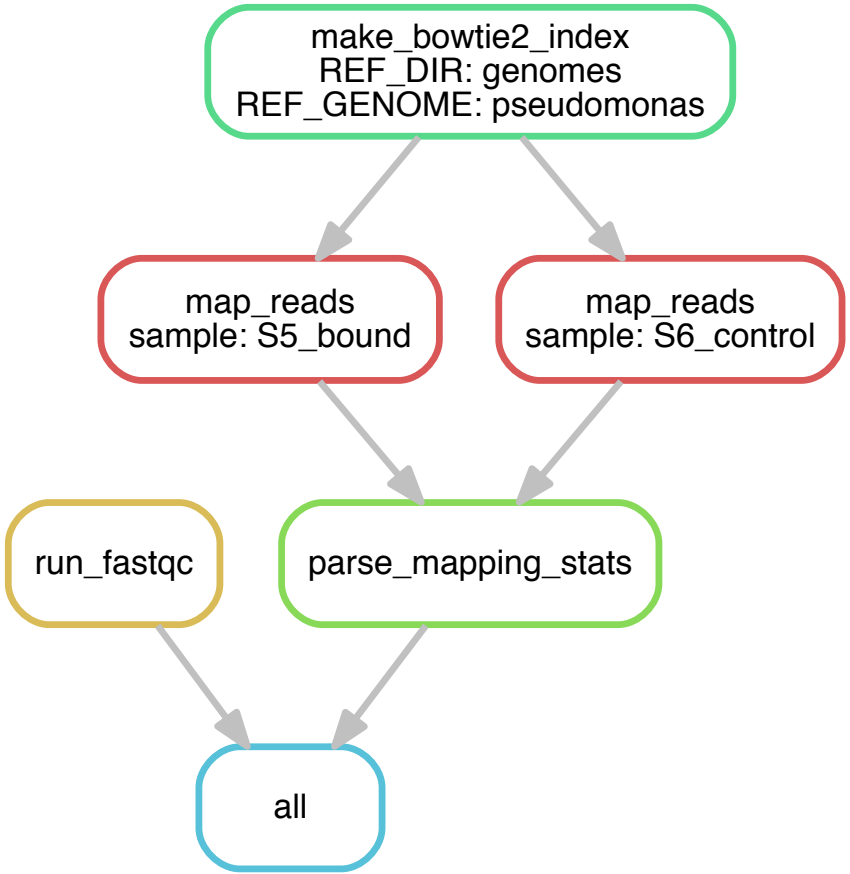
Rules all defines the output

```
rule all:
    input:
        # check quality
        expand("fastqc/{full_name}_fastqc.zip", full_name = full_names),
        # alignment stats summary
        "results/probes_alignment_stats.csv",
```

The listed files are the requested **output** from running the snakemake file

- Which other rule(s) can be run to produce these files?
- Traces back until it finds files that exist

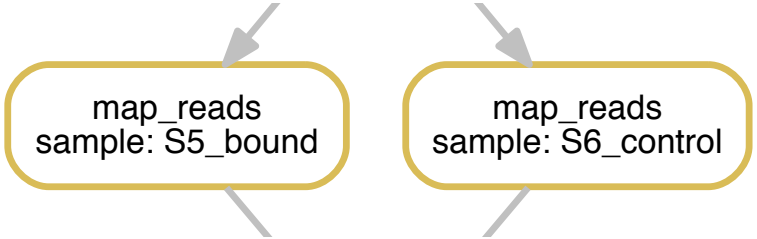
Rules all defines the output



Wildcards provide flexibility

```
rule map_reads:
    input:
        R1="samples/{sample}_L001_R1_001.fastq.gz",
        R2="samples/{sample}_L001_R2_001.fastq.gz",
    output:
        sam="aligned_reads/{sample}_aligned.sam",
```

- Inputs, outputs, log files must contains the same wildcards
- The same rule can be run multiple times with different wildcard values



Wildcards can be restricted

- A wildcards value is deduced from the file names
- You can constrain a wildcard's format to increase specificity
  - At file head or in a specific rule

Example:

```
input = {sample}.{group}
File name: A.1.normal.txt

Sample = "A.1" and group = "normal"
or
sample = "A" and group = "1.normal"
```

Constrain dataset:

```
input = {sample,[A-Z]+}.{group}
```

```
wildcard_constraints:
    sample = "[0-9]+",
    pre = "M|XT"
```

Expand recognizes multiple files at once

Instead of:

```
rule all:
    input:
        # check quality
        "S5_bound_L001_R1_001_fastqc.zip",
        "S5_bound_L001_R2_001_fastqc.zip",
        "S6_control_L001_R1_001_fastqc.zip",
        "S6_control_L001_R2_001_fastqc.zip",
```

Use:

```
rule all:
    input:
        # check quality
       expand("fastqc/{full_name}_fastqc.zip", full_name = full_names),
```

Expand allows running many samples at once using a list

Config allows listing samples

```
##----------------------------------------------##
## Config and Helper Files                      ##
##----------------------------------------------##
configfile: "config.yml"

# Get sample names (e.g. "S5")
verbose = config['verbose']
IDS, full_names = utils.load_samples(config, verbose=verbose)
```

- Reads a yml file in as a dictionary
- Include a list or dictionary of samples for easy scaling and reproducibility
- Python code is allowed in a snakemake file

Parts of a rule

- Input – files that need to exist for this rule to work
- Params – variables used in the rule
- Log – log file(s) produced by the rule
- Priority – how urgently a rule should be run
- Conda – environment to use for the rule
- Output – files produced by this rule
- Execution – the commands to get from input to output

Parts of a rule

```
rule parse_alignment_stats:
    input:
        bams=expand('aligned_reads/{sample}_aligned_sorted.bam', sample=IDS),
    params:
        bam_dir=directory("aligned_reads/"),
        log_dir=directory("log_files/"),
        extension='*aligned_stats.log',
        extension2='*_aligned_sorted.bam',
        out_fold='results/',
    log:
        "log_files/parse_alignment_stats.log"
    priority: 70
    conda:
        "envs/ggplot.yaml"
    output:
        stat="results/probes_alignment_stats.csv",
        read="results/probes_reads_per_chr.csv",
        rpm="results/probes_rpm_per_chr.csv",
        rpkm="results/probes_rpkm_per_chr.csv"
    shell:
        """
        (python summarize_alignment.py {params.bam_dir} {params.log_dir} \
          {params.out_fold} -d probes -lf {params.extension} -b {params.extension2} \
          -v {verbose} -p {REF_DIR}/{GENOME_NAME}) &>{log}
        """
```

Params

```
    params:
        bam_dir=directory("aligned_reads/"),
        log_dir=directory("log_files/"),
        extension='*aligned_stats.log',
        extension2='*_aligned_sorted.bam',
        out_fold='results/',
    shell:
        """
        (python summarize_alignment.py {params.bam_dir} {params.log_dir} \
          {params.out_fold} -d probes -lf {params.extension} -b {params.extension2} \
          -v {verbose} -p {REF_DIR}/{GENOME_NAME}) &>{log}
        """
```

- Useful for specifying non-files
- Directory required for input/output folders (not for params)
- Can include or not wildcards.
    - Commonly used for passing the sample description: `{sample}`

Log files

```
    log:
        "log_files/parse_alignment_stats.log"
    shell:
        """
        (python summarize_alignment.py {params.bam_dir} {params.log_dir} \
          {params.out_fold} -d probes -lf {params.extension} -b {params.extension2} \
          -v {verbose} -p {REF_DIR}/{GENOME_NAME}) &>{log}
        """
```

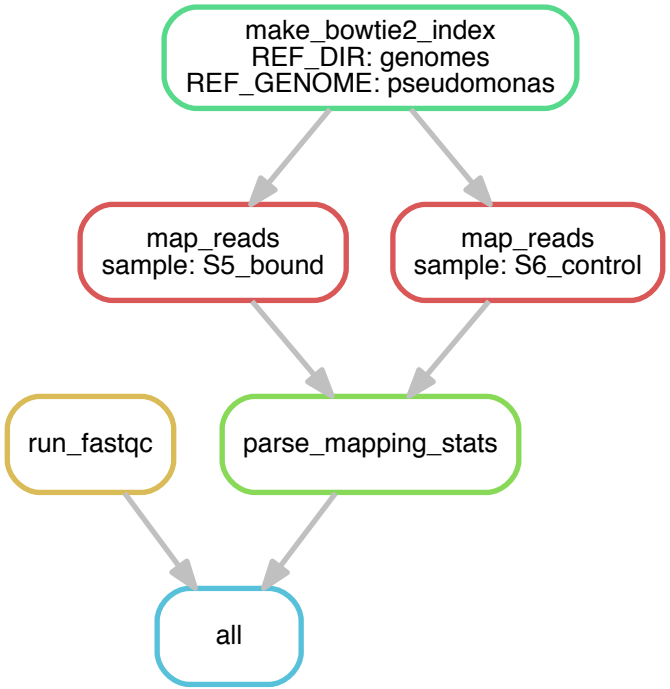
- Multiple named log files are allowed
- Can be passed directly to the program or bash can redirect the output
- Log files vs output
  - If a rule fails, snakemake deletes “output” files
  - Log files are kept for informational purposes

Temp files

Output files marked as temp are removed upon rule completion

```
rule map_reads:
    input:
        R1="samples/{sample}_L001_R1_001.fastq.gz",
        R2="samples/{sample}_L001_R2_001.fastq.gz",
        bt2_index = expand('{ref_dir}/{ref}.1.bt2', ref_dir=REF_DIR, ref=GENOME_NAME)
    output:
        sam=temp("aligned_reads/{sample}_aligned.sam"),
        bam = "aligned_reads/{sample}_aligned.bam",
        sorted = 'aligned_reads/{sample}_aligned_sorted.bam',
    log:
        "log_files/{sample}_aligned_stats.log"
    shell:
        """
        (bowtie2 -x {REF_DIR}/{GENOME_NAME} -p {core} -1 {input.R1} -2 {input.R2} -S {output.sam}) 2> {log}
        samtools view -bS {output.sam} > {output.bam}
        samtools sort {output.bam} -o {output.sorted}
        """
```

priority




```
    log:
        "log_files/parse_alignment_stats.log"
    priority: 70
    conda:
        "envs/ggplot.yaml"
```

- Used to control order of jobs that are available to be released at the same time
- Default is 0
- Useful when running multiple things at once (-j) or submitting to a cluster
  - Prioritize slower rules

Conda deployment

```
    conda:
        "envs/ggplot.yaml"
```

- Generally people try to run all rules in a single environment
- If that is not possible or not desired for some other reason, the specific conda environment to use can be specified
- Useful if I'm writing in python 3, but a package I want to use was written in python 2

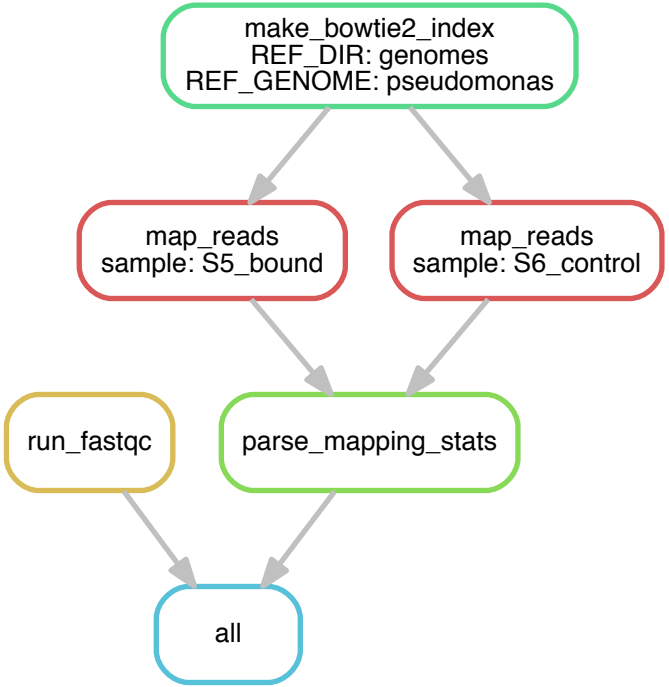
Parts of a rule

Snakecharm provides syntax highlighting for pycharm

- Imperfect, but useful

```
rule parse_alignment_stats:
    input:
        bams=expand('aligned_reads/{sample}_aligned_sorted.bam', sample=IDS),
    params:
        bam_dir=directory("aligned_reads/"),
        log_dir=directory("log_files/"),
        extension='*aligned_stats.log',
        extension2='*_aligned_sorted.bam',
        out_fold='results/',
    log:
        "log_files/parse_alignment_stats.log"
    priority: 70
    conda:
        "envs/ggplot.yaml"
    output:
        stat="results/probes_alignment_stats.csv",
        read="results/probes_reads_per_chr.csv",
        rpm="results/probes_rpm_per_chr.csv",
        rpkm="results/probes_rpkm_per_chr.csv"
    shell:
        """
        (python summarize_alignment.py {params.bam_dir} {params.log_dir} \
          {params.out_fold} -d probes -lf {params.extension} -b {params.extension2} \
          -v {verbose} -p {REF_DIR}/{GENOME_NAME}) &>{log}
        """
```

Running the code




```
# run the pipeline
snakemake -s Snakefile.smk --jobs 1
```

Specify what file to run and any other options

Dry run gives you a preview

command

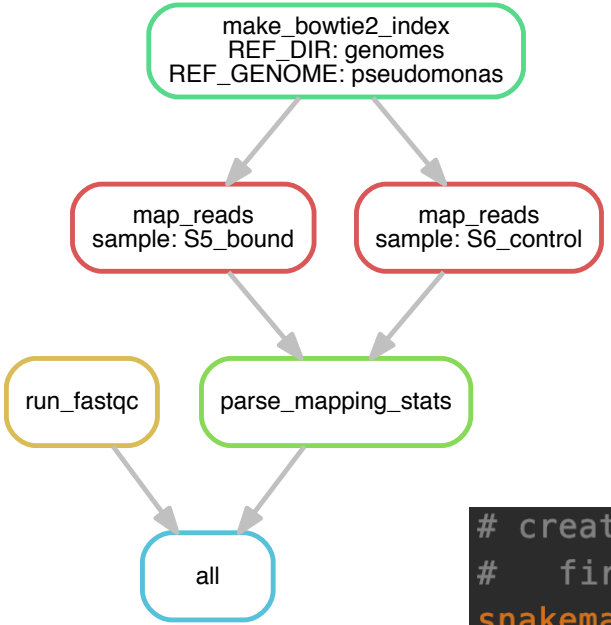
```
# activate the conda environment
conda activate snakemake_tutorial
# complete a dry run
snakemake -s Snakefile.smk -n -r
```

Partial output

```
Building DAG of jobs...
Job counts:
        count   jobs
        1       all
        1       make_bowtie2_index
        2       map_reads
        1       parse_mapping_stats
        1       run_fastqc
        6
```

- Prints which rules it will run, wildcard values, and why
- -n indicates dry run
- -r tells it to print the reason the rule was run

Directed Acyclic Graph



- Snakemake builds a directed acyclic graph (DAG) to determine rules to run
- Visualization helps understand steps and their connections

- fails (with a somewhat ambiguous message) if you have any print statements not in a rule

```
# create an image of the rules
#   first, within config.yml, set verbose to False
snakemake --forceall -s Snakefile.smk --dag | dot -Tpdf > dag.pdf
```

Running it!

```
Building DAG of jobs...
Using shell: /bin/bash
Provided cores: 12
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       all
        2       map_reads
        1       parse_alignment_stats
        1       run_fastqc
        5

[Wed Apr  1 11:27:32 2020]
rule map_reads:
    input: samples/S6_control_L001_R1_001.fastq.gz,
    output: aligned_reads/S6_control_aligned.sam, al
_sorted.bam
    log: log_files/S6_control_aligned_stats.log
    jobid: 4
    wildcards: sample=S6_control


[Wed Apr  1 11:27:32 2020]
rule map_reads:
    input: samples/S5_bound_L001_R1_001.fastq.gz, sa
```

Other options

- `--jobs`: the max number of rules/CPU cores that will be executed in parallel
- `--latency-wait 60`: allows 60 seconds for files to appear before a rule fails
- `-k`: keep going on independent jobs if one fails
- `-forceall`: execute all rules even if the file already exists
- -R map_reads: re-run map_reads and all downstream jobs

Resource specification

General resource options
- `-j 2`: run at most 2 jobs in parallel
- `--cores 4`: use 4 cores
- `--resources`: define other resources
  - CPU cores, GPU cores

Cluster resource specifications
- `--cluster`: the type of cluster you are submitting to and what resources to use

```
snakemake --snakefile Snakefile.smk -j 12 --cluster
"sbatch -n 1 -c 4 -p short --mem 4G -t 0-4:00
```

Resource specification (.json)

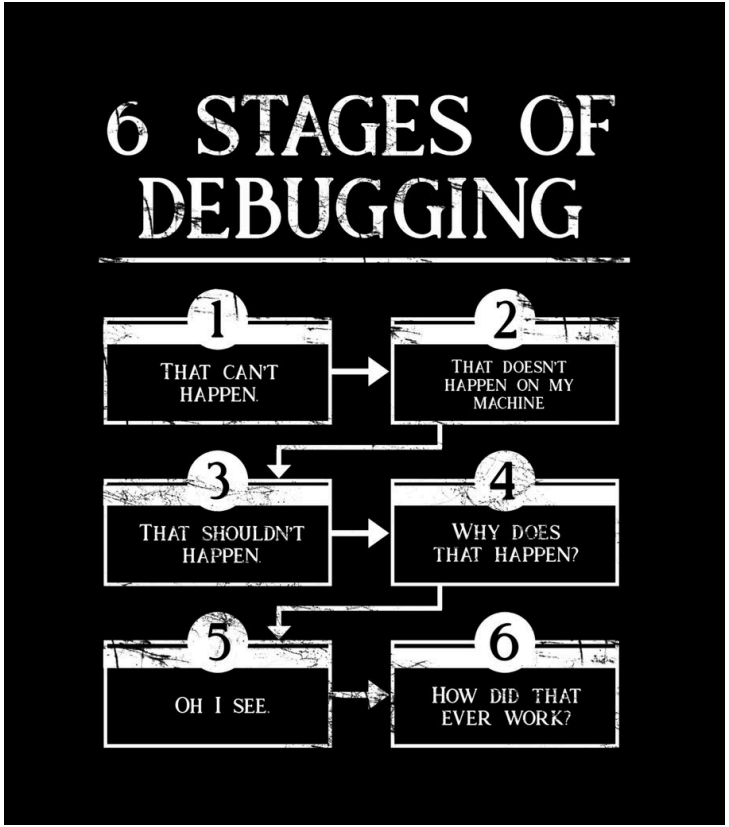
When not all rules need the same resources, a json can be used to specify rule specific allocations


```
snakemake --snakefile Snakefile.smk -j 12 --cluster-config
cluster.json --cluster "sbatch -n {cluster.n} -c {cluster.c}
-p short --mem {cluster.mem} -t {cluster.t}"
```

Debugging Tips

- Understanding wildcards is tricky.
  - If a dry run fails, this might be the reason
  - Snakemake does not know what you think the wildcard is

- Debug the command directly
  - When a rule fails, the log contains the exact command it attempted to run
  - Fix that, then adjust the rule to match

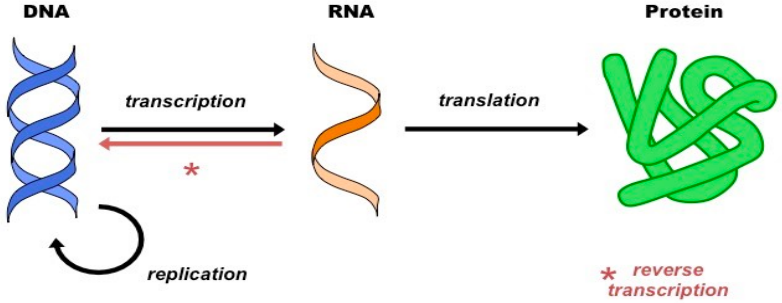


Overview

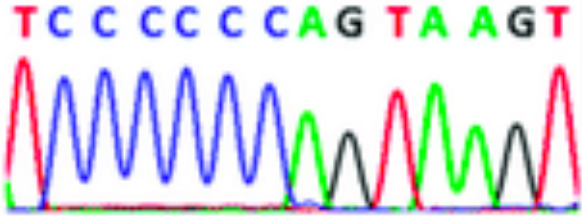
- Intro to Snakemake
- Snakemake Syntax
- **Snakemake example**
  - Quality control, alignment, and summarization of sequencing data
- Snakemake in Research: Detecting genetic engineering

Genetics 101

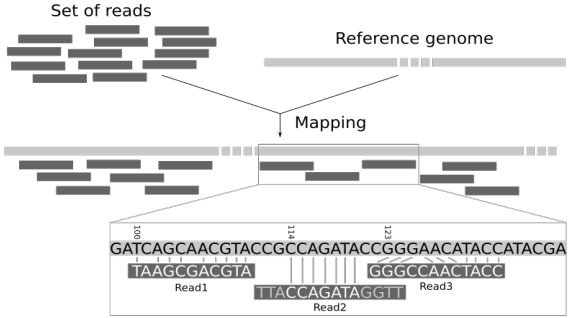
DNA provides the directions for everything alive



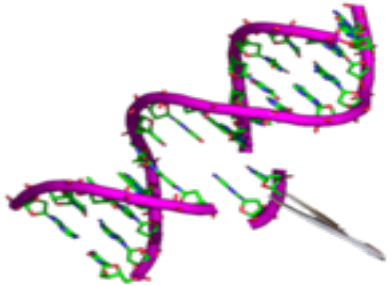
Sequencing is the process of reading nucleotides (ACTG) in DNA



Analysis includes mapping reads to a reference genome

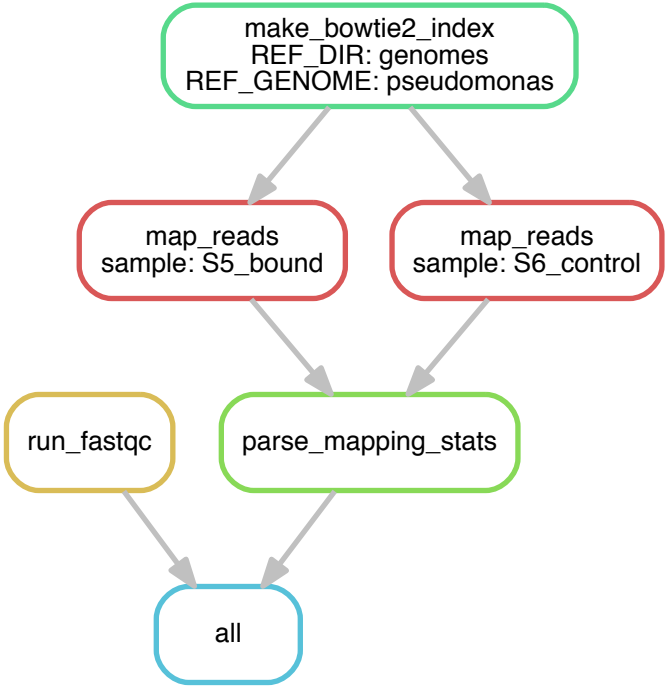


Genetic engineering manipulates the sequence (& function)



DRAPER

Quality control, alignment, and summarization



5 rules

- all
- make bowtie2 index
- map reads
- parse mapping stats
- run_fastqc

DRAPER

https://github.com/laseaman/odsc_snakemake_tutorial

DEMO

https://github.com/laseaman/odsc_snakemake_tutorial

Walking through the code in the repo

- README.md
- Snakefile.smk
  - utils and config
  - rule all
  - temp in map_reads
- config.yml
- utils.py
- snakemake_tutorial_env.yml
- summarize_alignment.py
- Run things
  - Dry run
  - Make DAG
  - Run
- Results: fastqc, alignment summary file

Take Aways

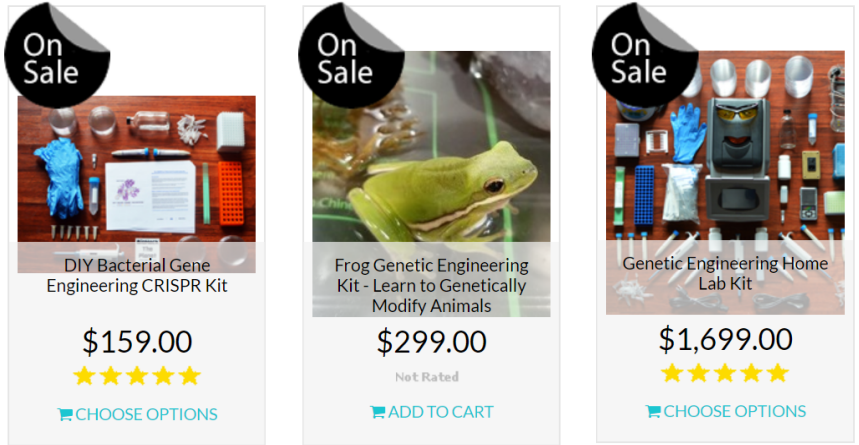
- Snakemake is a platform for creating repeatable, scalable data processing pipelines
- Rules are used to define the steps in the analysis through their input and output files
- Wildcards provide repeatability
- Easy cluster interfacing and resource management provides scalability

Overview

- Intro to Snakemake
- Snakemake Syntax
- Snakemake examples
- **Snakemake in Research: Detecting genetic engineering**

FELIX Detecting engineered organisms

CRISPR and other techniques enable modification and release of genetically engineered organisms

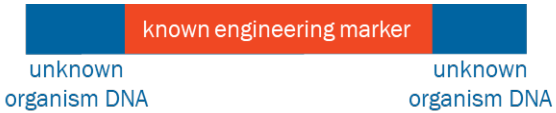


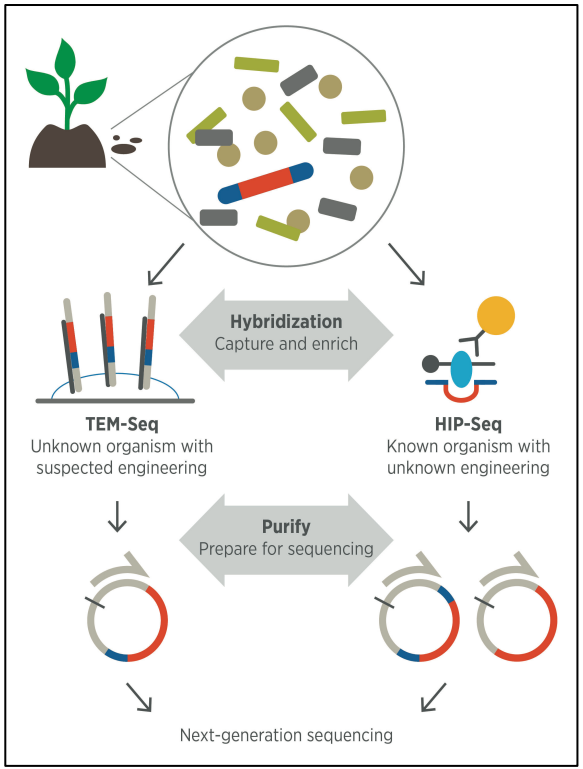
DIY genome editing tools are becoming widely available, with uncertain risks

A team with Draper, Harvard and BU is developing new tools for rapid identification of engineered organisms in complex environmental samples

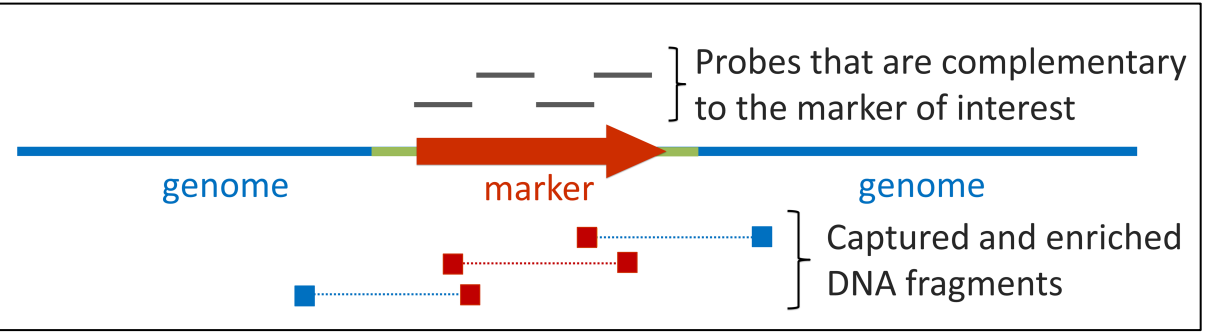
DRAPER

TEM-Seq and HIP-Seq: A two-pronged approach to enrich for engineered DNA



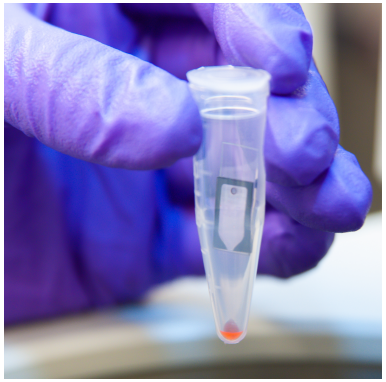


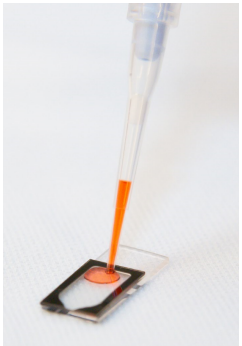
Enrichment Overview

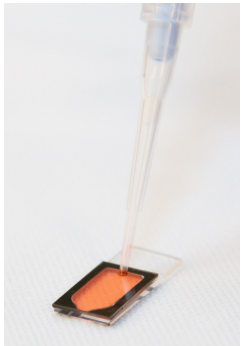


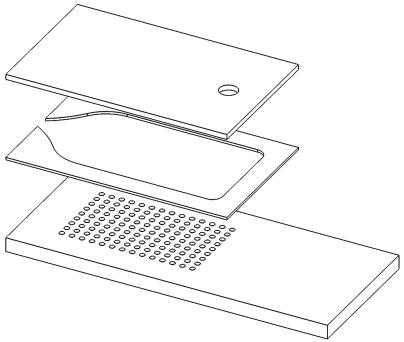
Number of markers	Class of marker	Number of probes
12	Promoters	34
1	Enhancer	3
6	Terminators	11
3	Fluorophores	15
9	Resistance markers	58
3	Affinity tags	5
2	Gene editing tools	16
2	Plasmid origins	12
4	Toxin surrogates	38

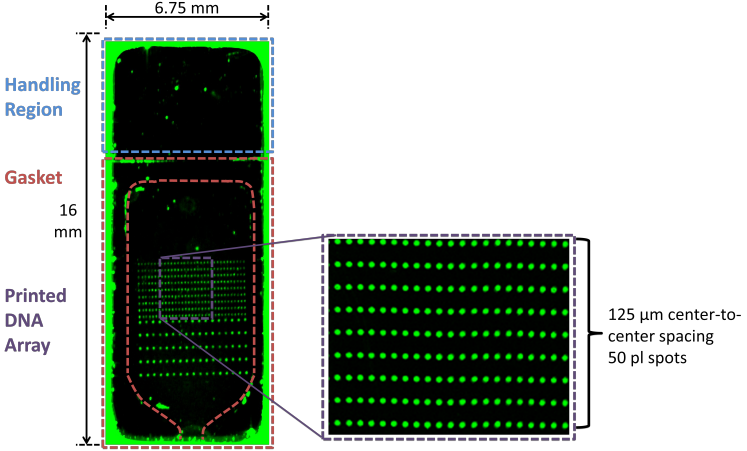
Mini-microarray: enabling low-volume processing & sample recovery







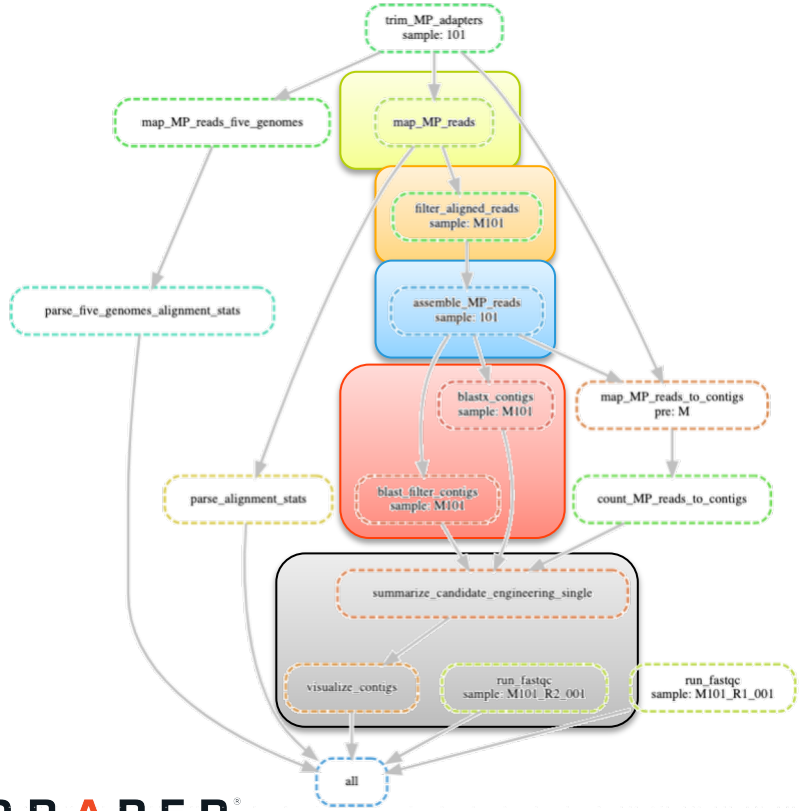




Specification	Mini Microarray	Commercial microarrays
Optical Detection	Yes	Yes
Sample Volume	**5.4 µl**	**>10 µl**
Sample Recovery	**Yes**	**No**
Number of Probes	~2,500 (current geometry)	Various
Customizability	**Yes**	**No**

DRAPER

Telling a story: Pipeline Overview



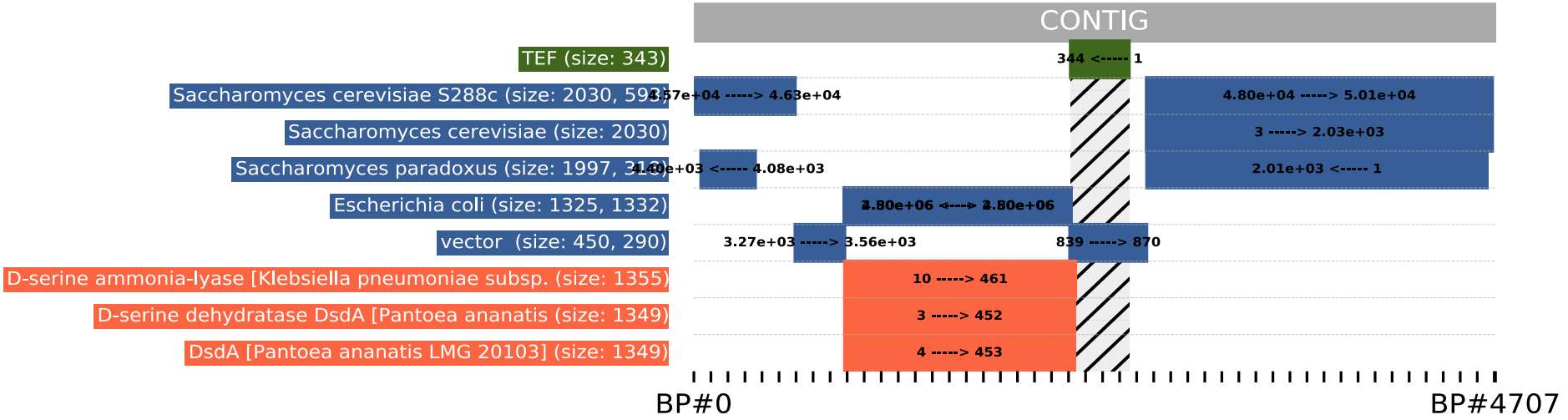
- Alignment
- Filtering
- Assembly
- Annotation
- Visualization/Summarization

DRAPER

Align to probes and filter reads

Assemble reads

Identify engineering (BLAST, visualization)



E. coli gene *dsdA* under control of the TEF promoter inserted into the *S. cerevisiae* HO locus

Snakemake in FELIX

Snakemake allows us to connect 10 different programs using 54 rules

We have processed 50 samples (~200 Gb) in 4 hours using a high performance cluster at Harvard

Challenges:

- Rule redundancy caused by different parameters for different types of samples
- Cluster integration causes inefficiencies because grouping doesn't work