PORPIDpipeline

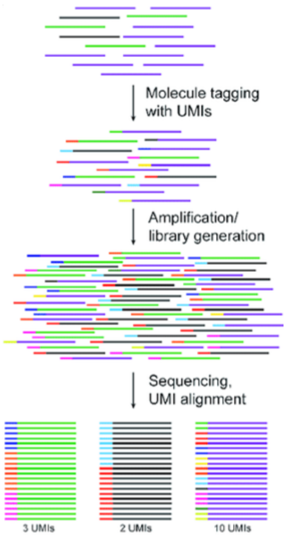
https://github.com/MurrellGroup/PORPIDpipeline

sponsored by
Division of Medical Virology,
Department of Pathology,
University of Cape Town.

2022

Outline

Introduction - UMI sequencing


Molecule tagging with UMIs
Amplification/ library generation
Sequencing, UMI alignment
3 UMIs
2 UMIs
10 UMIs

Unique Molecular Identifiers (UMIs) are short sequences incorporated into each fragment before PCR amplification so as to identify the molecule of origin for each read and to identify PCR duplicates.

A short (6bp) user selected UMI is affixed during cDNA synthesis and identifies the *donor*.

A longer (8bp) randomly generated UMI identifies duplicate reads originating from the same molecule, these are called *amplicons*.

Wetlab - reverse transcription

An RNA sample is obtained from a donor. The RNA is converted to cDNA through one round of reverse transcription using a cDNA primer composed of four parts. for example:

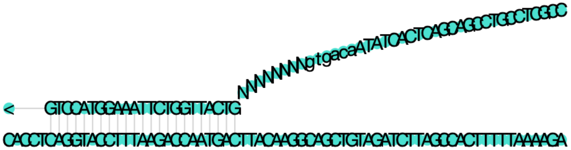
```
cDNA_primer: CCGCTCCGTCCGACGACTCACTATAacagtgNNNNNNNNGTCATTGGTCTTAAAGGTACCTG
```

1) first we have a 25nb receptor that will be used as a binding site in the PCR steps to follow
2) next we have a 6nb user selected donor barcode shown in lower case
3) next we have an 8nb random barcode which identifies the cDNA strand to be amplified
4) lastly we have a 23nb primer that will bind to the RNA upstream of a region of interest, henseforth referred to as the *amplicon* (that to be amplified).

Wetlab - cDNA

The cDNA primer is designed so that, on reversal, the portion after the N's will bind to the RNA strand upstream from the amplicon.

The remaining components of the reversed primer *hang* off the primer upstream of the binding site.


NNNNNNNgtgacaATATCACTCAGCAGCCTGCCTCGCC
<
GTCCATGGAAATTCTGGTTACTG
CACCTCAGGTACCTTTAAGACCAATGACTTACAAGGCAGCTGTAGATCTTAGCCACTTTTTAAAAGA

After binding the *reverse transcriptase* enzyme extends the reversed primer, thus complimenting the RNA to obtain cDNA.

for more details, watch this YouTube video.

Wetlab - PCR first round

After purification our cDNA fragments are moved to another instrument for PCR amplification. All fragment have identical donor barcodes but a unique randomly generated CDNA barcode.

Two primers are used for the PCR reactions, a new primer called the `sec_str_primer` and the first part of the previous `cDNA_primer`

```
sec_str_primer: TAGGCATCTCCT
cDNA_primer: CCGCTCCGTCCGACGACTCACTATA
```

In the first round of PCR, the `cDNA_primer` does no work but the `sec_string_primer` binds **downstream** of the amplicon in the cDNA.

Wetlab - PCR continued. . .

In the first round of PCR, the DNA polymerase extends the `sec_string_primer` complementing the cDNA strand to the end of the `cDNA_primer`.

From the second round onwards the `cDNA_primer` binds upstream of the donor's barcode and both primers are extended.

If we start the PCA reaction with N distinct cDNA fragments, each with a unique random barcode, then, assuming successful primer extensions, round by round amplification will proceed as follows

$$N \xrightarrow{round(1)} N \xrightarrow{round(2)} 2N \xrightarrow{round(3)} 4N \cdots \xrightarrow{round(r)} 2^{r-1}N$$

for an explanation of the PCR process watch this YouTube video.

Wetlab - PacBio

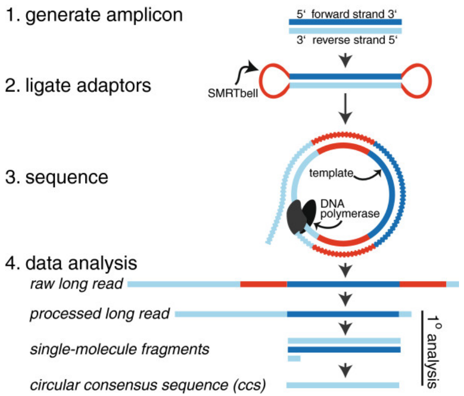
At the end of the PCR process we have many copies of DNA strands from the same donor with each strand incorporating both a user specified donor barcode and a random cDNA barcode from which the strand was amplified.

Samples from different donors and different amplicons are collected for one PacBio sequencing run.

The PacBio system attaches a circular ligate adaptor at each end of the DNA fragment and then sequencing begins by means of a primer that binds to the ligate and extends many times around the fragment issuing sequencing signals for each extension base.

After the sequencing has completed PacBio software performs sequence assembly and returns the sequences in `fastq` format which gives both the sequence bases and a confidence measure for each base sequenced (see next slide).

Wetlab - PacBio . . .



for further elucidation watch this YouTube video.

Snakemake - introduction

Snakemake is a workflow management system for pipelines that handles:

- suspension and resumption
- logging
- data origins
- parallelization

To use Snakemake, you must decompose workflow into rules.

- rules define how to obtain output files from input files.

Snakemake will infer dependencies and execution order from a set of rules stored in a `snakefile`.

Snakemake - rules

- rules are *named*
- rules are *indented*
- there can be *multiple named* input and output files.
- names can incorporate *wildcard* parts
- a rule has a *shell script* telling how to generate output from input.

```
rule sort_and_annotate:
    input:
        a="path/to/{dataset}.txt",
        b="path/to/annotation.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "paste <(sort {input.a}) {input.b} > {output}"
```

Snakemake - jobs

Dependencies are determined top-down.

- For a given target, a rule that can be applied to create it is called a *job*.
- For the input files of the job, rules are determined recursively.

A job is executed if and only if

- output file is target and does not exist
- output file needed by another executed job and does not exist
- input file newer than output file
- input file will be updated by other job
- execution is enforced by command line parameters

determined via breadth-first-search of the Directed acyclic graph (DAG) of the workflow.

Snakemake - command line

assuming that the workflow is defined in `snakefile`.

```
# execute the workflow with target D1.sorted.txt
snakemake D1.sorted.txt

# execute the workflow without target: first rule defines target
snakemake

# dry-run
snakemake -n

# dry-run, print shell commands
snakemake -n -p

# dry-run, print execution reason for each job
snakemake -n -r

# visualize the DAG of jobs using the Graphviz dot command
snakemake --dag | dot -Tsvg > dag.svg
```

Snakemake - example

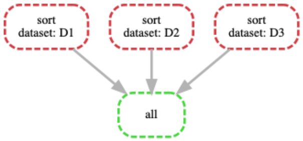
contents of `snakefile`

```
DATASETS = ["D1", "D2", "D3"]

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=DATASETS)


rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "sort {input} > {output}"
```

workflow DAG



Snakemake - configfile

Often you want your workflow to be customisable, so that it can easily be adapted to new data. For this purpose, Snakemake provides a config file mechanism. Config files are specified with the configfile directive. Snakemake will load the config file and store its contents into a globally available *dictionary* named `config`.

`config.yaml`

```
datasets:
  D1:
  D2:
  D3:
```

`snakefile`

```
configfile: "config.yaml"

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=config["datasets"])

rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "sort {input} > {output}"
```

to continue . . .

For the remainder of this presentation we will describe each script in PORPIDpipeline and show how to invoke the script via SnakeMake.

At the beginning of the `snakefile` we have the following instructions to parse the `demo` PORPIDpipeline config file shown on the next slide.

```
configfile: "config.yaml"
DATASETS = [d for d in config for s in config[d]]
SAMPLES = [s for d in config for s in config[d]]
```

In the demo config file we have one dataset, `demo`, in which there are 6 samples (3 donors each with 2 amplicons).

For each sample we specify the primers used and a panel file.

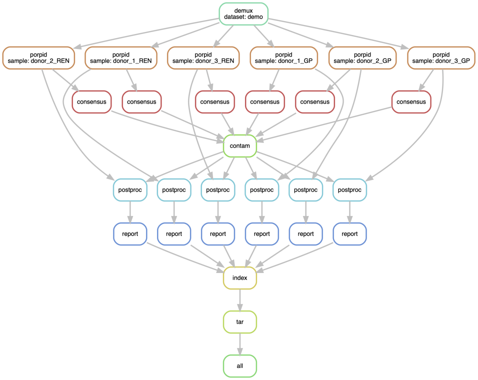
to continue . . . . . .

demo PORPIDpipeline `config.yaml`:

```
demo:
  donor_1_REN:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAacagtgNNNNNNNNGTCATTGGTCTTAAAGGTACCTG
    sec_str_primer: TAGGCATCTCCT
    panel: "panels/HIV1_COM_2017_5970-8994_DNA_stripped.fasta"
  donor_2_REN:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAcactcaNNNNNNNNGTCATTGGTCTTAAAGGTACCTG
    sec_str_primer: TAGGCATCTCCT
    panel: "panels/HIV1_COM_2017_5970-8994_DNA_stripped.fasta"
  donor_3_REN:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAggtagcNNNNNNNNGTCATTGGTCTTAAAGGTACCTG
    sec_str_primer: TAGGCATCTCCT
    panel: "panels/HIV1_COM_2017_5970-8994_DNA_stripped.fasta"
  donor_1_GP:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAacagtgNNNNNNNNGTATGTCATTGACAGTCCAGC
    sec_str_primer: TTGACTAGCGGAGGCTAGAAGGAGA
    panel: "panels/HIV1_COM_2017_787-3300_DNA_stripped.fasta"
  donor_2_GP:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAcactcaNNNNNNNNGTATGTCATTGACAGTCCAGC
    sec_str_primer: TTGACTAGCGGAGGCTAGAAGGAGA
    panel: "panels/HIV1_COM_2017_787-3300_DNA_stripped.fasta"
  donor_3_GP:
    cDNA_primer: CCGCTCCGTCCGACGACTCACTATAggtagcNNNNNNNNGTATGTCATTGACAGTCCAGC
    sec_str_primer: TTGACTAGCGGAGGCTAGAAGGAGA
    panel: "panels/HIV1_COM_2017_787-3300_DNA_stripped.fasta"
```

to continue . . . .

The image below is a directed acyclic graph (DAG) of the pipeline showing all the scripts and the order of execution



In the following slides we discuss each script in the pipeline.

demux

Because PacBio runs are expensive, samples are tagged and pooled for one sequencing run. This is called *multiplexing* and after the sequencing, the sequences generated must be *de-multiplexed*.

This *de-multiplexing* allows us to separate the full dataset into sample datasets which are more manageable in size and which can be further processed in parallel.

In the demux script we read the pac-bio dataset in *chunks* directly from a zipped file and then we perform quality filtering and de-multiplexing on each chunk and then we append each de-multiplexed sequence to the appropriate demux file for that sample.

demux - quality filtering

Raw reads from a PacBio sequencing run comprise of DNA fragments with confidence scores for each nucleotide in the sequence. For example:

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

The confidence scores, also known as *Phred* quality scores, are integers from 0 to 93, encoded using ASCII characters 33 to 126, and a log scale on the probability of an incorrect call.

To compute the probability P of an incorrect base call we evaluate

$$P_{incorrect} = 10^{-\frac{Q-33}{10}}$$

for each ASCII character Q in the quality string.

demux - quality filtering . . .

In our quality filter we reject fragments who's mean probability of an incorrect call (error rate) is greater than a user supplied parameter.

We also reject fragments who's length is either too small or too large as specified by user supplied parameters.

In our PorpidPostroc H704 pipeline we use the following values for these parameters.

error_rate = 0.01
min_length = 2100
max_length = 4000

demux - de-multiplexing

Recall that DNA was selected for amplification through the use of primers. For each donor/amplicon combination two primers are specified in a *config* file for the pool.

In the `demux` script we parse the config file and using the donor barcodes from the `cDNA_primer` primer we are able to group the `fastq` filtered reads by *donor*.

Then using the `sec_str_primer` we are able to sub-group each group of donor reads by *amplicon*.

demux - snakefile

```
rule demux:
    input:
        "raw-reads/{dataset}.fastq.gz"
    output:
        directory("porpid/{dataset}/demux"),
        "porpid/{dataset}/quality_report.csv",
        "porpid/{dataset}/demux_report.csv"
    params:
        chunk_size = 10000,
        error_rate = 0.01,
        min_length = 2100,
        max_length = 4000,
        config = lambda wc: config[wc.dataset]
    script:
        "scripts/demux.jl"
```

demux - snakefile . . .

The first `snakemake` rule in PORPIDpipeline is for the `demux` script which in our case loads the PacBio raw reads from `raw-reads/demo.fastq.gz` in *chunks*, applies quality filter and demux methods to each chunk, and then appends the de-multiplexed output sequences to the appropriate file in the output directory `porpid/demo/demux/`.

In the rule, the string {`dataset`} is replaced by the string `demo`, from the top level of the config dictionary.

Quality filter parameters are passed to the script using the parameter block.

In order to de-multiplex, the script must have access to the primers used for each sample. This is achieved by passing the whole the `config` file to the demux script as a Snakemake parameter.

porpid

After the `demux` script has run the PacBio reads have been grouped according to donors and amplicons and each group is stored in its own file. However in the sample, before PCR takes place, there may be many different instances of a particular donor/amplicon sequence. Each instance will have a unique *random* barcode assigned and after PCR amplifications there will be many copies of this sequence generated.

The purpose of the `porpid` script is to further sub-group the donor/amplicon files by random barcode identifiers. To this end each donor/amplicon file is read, a list of random barcodes found is generated and for each random barcode found a new sub-group, *donor/amplicon/barcode*, file is output.

what could go wrong with this plan, see next slide . . .

porpid . . .

PacBio reads are subject to errors and these errors could occur in reading the randomly generated barcode. Barcodes with fewer than 5 CCS reads are filtered out. However barcodes with more than 5 CCS may still be defective and we must either reject the sequences or we must assign the sequences to a collection of *nearby* sequences. In this context *nearby* refers to sequences with barcodes similar to our defective barcode.

This detection and reassignment of defective barcodes is carried out using a statistical technique from the field of *natural language processing* called the Latent Dirichlet Allocation (LDA).

LDA is a generative statistical model that allows sets of observations to be explained by unobserved groups. Given an observed barcode, the `porpid` script makes use of LDA to allocate a sequence to a *most likely barcode group*.

porpid . . . . . .

The `porpid` script also makes use of the quality scores to filter out so-called *heteroduplexes*.

This needs further explanation . . .

porpid - snakefile

```
rule porpid:
    input:
        "porpid/{dataset}/demux"
    output:
        directory("porpid/{dataset}/porpid/{sample}.fastq"),
        "porpid/{dataset}/tags/{sample}.csv"
    params:
        config = lambda wc: config[wc.dataset][wc.sample]
    script:
        "scripts/porpid.jl"
```

The porpid script loads the de-multiplexed `samples` and groups them according to random barcode which it uses as a filename, when writing to the output directory.

This is a directory (not a file) and the extension `.fastq` is probably superfluous.

Note that this time the `sample` strings are passed to the porpid script through the global `params` dictionary which the script accesses using the code:

```
config = snakemake.params["config"]
```

consensus

At this stage we have many reads representing each *donor/amplicon/barcode*. These should align perfectly but differences can occur and a *consensus* must be taken at each nucleotide position to eliminate errors in the PacBio processes.

Consensus sequences are generated from each *likely real* `fastq` file using *kmer* vector clustering and refinement packaged in `RobustAmpliconDenoising.jl`

Read agreement at each position is tracked using kmer seeded pairwise alignment to the candidate consensus. The minimal agreement value is reported for each consensus sequence.

consensus - snakefile

```
rule consensus:
    input:
        "porpid/{dataset}/porpid/{sample}.fastq"
    output:
        "porpid/{dataset}/consensus/{sample}.fasta"
    params:
        config = lambda wc: config[wc.dataset][wc.sample]
    script:
        "scripts/consensus.jl"
```

The input, `porpid/dataset/porpid/sample.fastq` is in fact a directory consisting of `fastq` collections for each random barcode. The script performs a consensus for each of these generating one `fasta` record.

These consensus sequences are collected for each sample and stored in the output file:
`porpid/dataset/consensus/sample.fasta`

the sample names are obtained from the config file which again is passed to the script as a Snakemake parameter.

contam

All consensus sequences for the same sample (donor + amplicon) are subjected to kmer vector clustering using a 1.5% radius.

Clusters making up more than 20% of the sample population are merged with a cluster representing the mean of all clusters for that sample.

All sample clusters found above are merged with kmer representations of common lab contaminants that must stored by the user in the `contam_panel.fasta` file.

This clustering allows for sequence *winnowing*, either because of sequence *leakage* from one cluster to another within the sample or else because the the sequence in question is a *contaminant* derived from one of the suspected contaminating sequences.

contam . . .

Sequences that are within a 1.5% corrected kmer vector distance from another cluster centroid AND $>$ 1.5% away from their own cluster centroid are discarded. A message is copied to the `contam_check.csv` file.

Sequences that are within a 1.5% corrected kmer vector distance from another cluster centroid AND $<$ 1.5% away from their own cluster centroid are retained but with a warning. A message is copied to the `contam_check.csv` file.

This method only is exposed to templates that are included in the same PacBio run with the intention of catching index hopping events, and does not replace a more rigorous BLAST search prior to final analysis.

For further information on kmer clustering please read this article.

contam - snakefile

The `contam` filter is run once and should be able to use **all** the consensus files to detect leakage (even across samples) and/or contamination. To allow for this we need one input that resolves to the list of consensus files.

To solve this problem Snakemake allows rules to access Python like Snakemake functions. In this case the function we are after is as follows:

```
def contam_input(wildcards):
    SAMPLES = [s for s in config[wildcards.dataset]]
    return expand("porpid/{dataset}/consensus/{sample}.fasta",
        dataset = wildcards.dataset,
        sample = SAMPLES
    )
```

In our example `wildcards.dataset` will resolve to `demo` and `SAMPLES` will resolve to our list of 6 samples in the config file.

contam - snakefile . . .

with this function in our armoury we can use it in the contam filter:

```
rule contam:
    input:
        files = contam_input,
        panel = "panels/contam_panel.fasta"
    output:
        directory("porpid/{dataset}/contam_passed"),
        directory("porpid/{dataset}/contam_failed"),
        "porpid/{dataset}/contam_report.csv"
    script:
        "scripts/contam.jl"
```

All consensus files are passed as input to **one** execution of the `contam_filter` script. See the DAG to verify this.

postproc

Sequences that survived the contamination check are aligned to a reference sequence using mafft and filtered to per-base pairwise consensus agreement of $\geq 70\%$

Insertions at the ends of the sequences are trimmed off via profile alignment to a reference panel

Major misalignments such as off target seqs ($\geq 50\%$ diff from profile of panel file) are excluded and pushed to `postproc/<sample_id>.fasta.rejected.fasta` .

This file is usually empty, but should be inspected by the user for confirmation of possible contamination.

postproc . . .

The sequences that survived the contamination filter are loaded and aligned against a reference sequence with file name passed as a Snakemake parameter.
A paired maximum likelihood tree (FastTree) and highlighter plot is generated from the aligned templates.

A 2D projection of the template sequences by multidimensional scaling, coloured by the probability of an inflated $G > A$ mutation rate using a nucleotide substitution model and overall consensus

this projection is a proxy for APOBEC hypermutation.

This needs further explanation . . .

postproc snakefile

```
rule postproc:
    input:
        "porpid/{dataset}/contam_passed",
        "porpid/{dataset}/tags/{sample}.csv"
    output:
        report("postproc/{dataset}/{sample}/{sample}.fasta.mds.png",
                    category = "postproc", caption = "report-rst/mds.rst"),
        "postproc/{dataset}/{sample}/{sample}.fasta.apobec.csv",
        report("postproc/{dataset}/{sample}/{sample}.fasta.tre.svg",
                    category = "postproc", caption = "report-rst/highlighter.rst"),
        "postproc/{dataset}/{sample}/{sample}.fasta",
        report("postproc/{dataset}/{sample}/{sample}_qc_bins.png",
                    category = "postproc", caption = "report-rst/bins.rst"),
        "postproc/{dataset}/{sample}/{sample}_qc_bins.csv",
        "postproc/{dataset}/{sample}/{sample}.fasta.rejected.fasta",
        "postproc/{dataset}/{sample}/{sample}.fasta.rejected.csv"
    params:
        panel = lambda wc: config[wc.dataset][wc.sample]["panel"]
        fs_thresh = 5,
        agreement_thresh = 0.7,
        panel_thresh = 50
    script:
        "scripts/postproc.jl"
```

postproc snakefile . . .

Note that the `postproc` script also requires input from `tags` data output from the `porpid` script, hence the curved edge in the pipeline DAG from three levels back.

Various images are produced and stored in output files for the next `report` script.

report

Reports are generated by combining ‘postproc‘ plots with explanations into an HTML page for each sample.

These reports pages include

- UMI reject table
- UMI strip-plot
- 2D MDS plot of likely-reals
- Phylogeny/Highlighter combo plot
- Blast table for major clades

report - UMI reject table

The first part of the sample report is a table of all sequences rejected by the Porpid script.

`BPB-rejects` sequences that were discarded due to bad primer blocks.

`LDA-rejects` likely offspring from other UMI bins.

`fs<5` indicates sequencing depth was under 5 CCS and too low for consensus analysis.

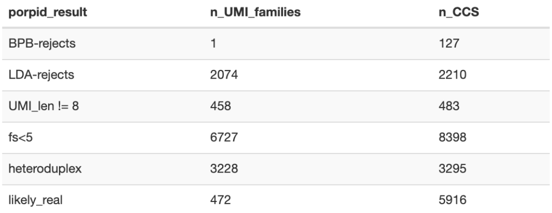
`heteroduplex` indicative of a superimposed signal of two different UMI sequences during circular consensus generation.

`UMI_len != 8` UMIs of length other than 8 are excluded from analysis.

`likely_real` sequences that pass the Porpid tests and are kept for downstream analysis.

report - UMI reject table - example

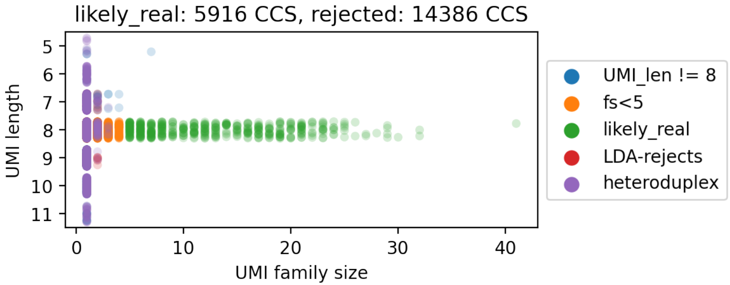
The table is an example of a typical reject table and shows the reason for the reject, the number of UMI families involved and the number of sequences rejected.



porpid_result	n_UMI_families	n_CCS
BPB-rejects	1	127
LDA-rejects	2074	2210
UMI_len != 8	458	483
fs<5	6727	8398
heteroduplex	3228	3295
likely_real	472	5916

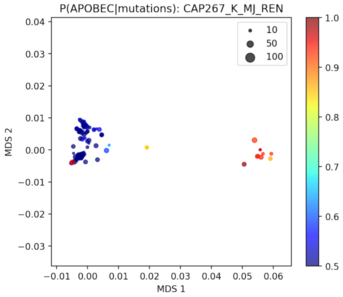
report - UMI strip-plot

A strip-plot is a dithered scatter plot of UMI families showing ‘UMI length‘ versus ‘UMI family size‘ and colored by ‘reject‘ reason. This is a nice way to highlight the rejects from the table on the previous slide.



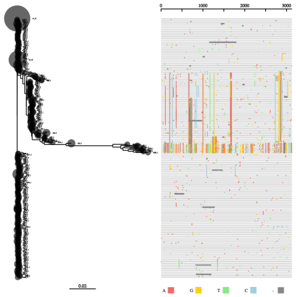
report - MDS plot of likely-reals

Postproc analysis of likely real sequences begins with a Multi-dimensional scaling projection into 2 dimensions in an attempt to show the major clades present in the sample.



report - Phylogeny/Highlighter of collapsed likely-reals

In this image identical sequences are collapsed into one representative and a phylogeny is constructed and displayed alongside a multiple alignment of sample sequences.



report snakefile

```
rule report:
    input:
        "postproc/{dataset}/{sample}/{sample}_qc_bins.png",
        "postproc/{dataset}/{sample}/{sample}_qc_bins.csv",
        "postproc/{dataset}/{sample}/{sample}.fasta.mds.png",
        "postproc/{dataset}/{sample}/{sample}.fasta.tre.svg",
        "postproc/{dataset}/{sample}/{sample}.fasta",
        "postproc/{dataset}/{sample}/{sample}.fasta.rejected.fasta",
        "postproc/{dataset}/{sample}/{sample}.fasta.rejected.csv"
    params:
        VERSION = VERSION,
        COMMIT = COMMIT,
        blast = "true",
        thresh_hold = "0.05",
        max_clades = "5",
        max_waits = "10"
    output:
        "postproc/{dataset}/{sample}/{sample}-report.html",
        "postproc/{dataset}/{sample}/{sample}-blast.csv"
    script:
        "scripts/report.jl"
```

Output from the `postproc` script is fed into the `report` script and `html` reports are generated and annotated with explanatory text and version identiiers. blast parameters are passed in the ‘params‘ block

index

This script generates an ‘html‘ index page for the individual sample report pages.

A rejection count summary table for all samples is also presented to convince the user that all sequences are accounted for.

index snakefile

```
rule index:
    input:
        expand("postproc/{dataset}/{sample}/{sample}-report.html", zip, dataset = DATASETS, sample = SAMPI
        expand("postproc/{dataset}/{sample}/{sample}-blast.csv", zip, dataset = DATASETS, sample = SAMPLES
    params:
        VERSION = VERSION,
        COMMIT = COMMIT,
        SAMPLES = SAMPLES
    output:
        "postproc/{dataset}/report_index.html",
        "postproc/{dataset}/{dataset}-blast.html"
    script:
        "scripts/index.jl"
```

Output from the `report` script is fed into the `index` script and an `html` index is generated and annotated with a summary of rejection counts.

tar

This script archives and compresses the `porpid` and `postproc` output directories to make it easy for the user to download results from a PORPIDpipeline server.

```
rule tar:
    input:
        "postproc/{dataset}/report_index.html"
    output:
         "porpid/{dataset}-porpid.tar.gz",
        "postproc/{dataset}-postproc.tar.gz"
    params:
        degap = "false",
        datasets = DATASETS,
        samples = SAMPLES
    script:
        "scripts/tar.jl"
```

The tar script waits for the index script to complete and then performs the archiving and zipping process with the help of a Julia script.

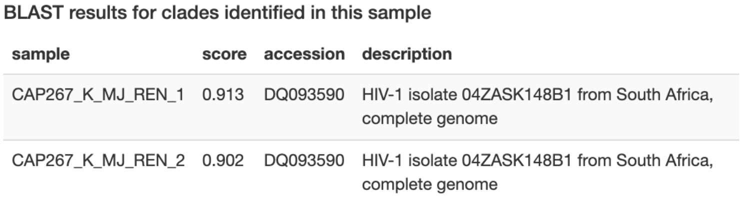
experimental BLAST pipeline

Finally, a second pipeline is provided for producing BLAST reports on each sample. If this pipeline is run (after the PORPIDpipeline) than a BLAST report is generated for representatives of major clades in each sample.

If a sample has rejected sequences then the longest sequence in the rejected file is appended to the BLAST query.

Note that the BLAST pipeline may have to be re-run many times in order to generate reports for every sample. This is due to the transient nature of the public BLAST server and users that rely on this BLAST facility may want to set up their own dedicated server.

example BLAST output



BLAST results for clades identified in this sample

sample	score	accession	description
CAP267_K_MJ_REN_1	0.913	DQ093590	HIV-1 isolate 04ZASK148B1 from South Africa, complete genome
CAP267_K_MJ_REN_2	0.902	DQ093590	HIV-1 isolate 04ZASK148B1 from South Africa, complete genome

a BLAST report that shows the top hit for each of two clades present in the sample.

Installation

The `PorpidPostroc` repository is located at:

https://github.com/MurrellGroup/PORPIDpipeline

This repository has quick start instructions on how to get PORPIDpipeline up and running in a minimal environment:

- installing third party software, `snakemake`, `mafft`, `fasttree`.
- cloning the PORPIDpipeline repository
- Installing Julia 1.7
- setting up the Julia environment for PORPIDpipeline.
- processing the `demo` data files packaged with PORPIDpipeline.

The quick start instructions also point the user to a script that allows full installation of PORPIDpipeline in a *Conda* environment.

Credits

This introduction to *PORPIDpipeline* and the individual script descriptions are based on the PORPID-postproc (branch h704) README written by *Alec Pankow*.

The introduction to *Snakemake* is based on introductory slides written by *Johannes Koester*, the originals of which can be viewed at http://slides.com/johanneskoester/deck-1.