dynast

Release 1.0.1

Kyung Hoi (Joseph) Min

Jul 01, 2022

INTRODUCTION:

CHAPTER

ONE

GETTING STARTED

Welcome to dynast!

Dynast is a command-line pipeline that preprocesses data from metabolic labeling scRNA-seq experiments and quantifies the following four mRNA species: unlabeled unspliced, unlabeled spliced, labeled unspliced and labeled spliced. In addition, dynast can perform statistical estimation of these species through expectation maximization (EM) and Bayesian inference. Please see *Statistical estimation* for more details on how the statistical estimation is performed.

1.1 Installation

The latest stable release of dynast is available on the Python Package Index (Pypi) and can be installed with pip:

```
pip install dynast-release
```

To install directly from the Git repository:

```
pip install git+https://github.com/aristoteleo/dynast-release
```

To install the latest *development* verson:

```
pip install git+https://github.com/aristoteleo/dynast-release@devel
```

Please note that not all features may be stable when using the development version.

1.2 Command-line structure

Dynast consists of four commands that represent four steps of the pipeline: `ref`, `align`, `consensus`, `count`, `estimate`. This modularity allows users to add additional preprocessing between steps as they see fit. For instance, a user may wish to run a custom step to mark and remove duplicates after the `align` step.

Command	Description
`ref`	Build a STAR index from a reference genome FASTA and GTF.
`align`	Align FASTQs into an alignment BAM.
`consensus`	Call consensus sequence for each sequenced mRNA molecule.
`count`	Quantify unlabeled and labeled RNA.
`estimate`	Estimate the fraction of labeled RNA via statistical estimation.

1.3 Basic usage

Prerequisites:

- FASTQ files from a metabolic labeling scRNA-seq experiment
- **[Optional]** STAR genome index for the appropriate organism. Skip the first step if you already have this.

1.3.1 Build the STAR index

First, we must build a STAR index for the genome of the organism that was used in the experiment. For the purpose of this section, we will be using the mouse (Mus musculus) as an example. Download the **genome (DNA) FASTA** and **gene annotations GTF**. If you already have an appropriate STAR index, you do not need to re-generate it and may skip to the next step.

```
dynast ref -i STAR Mus_musculus.GRCm38.dna.primary_assembly.fa.gz Mus_musculus.GRCm38.
↪102.gtf.gz
```

where `STAR` is the directory to which we will be saving the STAR index.

1.3.2 Align FASTQs

Next, we align the FASTQs to the genome.

```
dynast align -i STAR -o align -x TECHNOLOGY CDNA_FASTQ BARCODE_UMI_FASTQ
```

where `align` is the directory to which to save alignment files, and `TECHNOLOGY` is a scRNA-seq technology. A list of supported technologies can be found by running `dynast --list`. `BARCODE_UMI_FASTQ` is the FASTQ containing the barcode and UMI sequences, whereas the `CDNA_FASTQ` is the FASTQ containing the biological cDNA sequences.

1.3.3 [Optional] Consensus

Optionally, we can call consensus sequences for each sequenced mRNA molecule.

```
dynast consensus -g Mus_musculus.GRCm38.102.gtf.gz --barcode-tag CB --umi-tag UB -o␣
↪consensus align/Aligned.sortedByCoord.out.bam
```

where `consensus` is the directory to which to save the consensus-called BAM. Once the above command finishes, the `consensus` directory will contain a new BAM file that can be used as input to the following step.

1.3.4 Quantify

Finally, we quantify the four RNA species of interest. Note that we re-use the gene annotations GTF.

```
dynast count -g Mus_musculus.GRCm38.102.gtf.gz --barcode-tag CB --umi-tag UB -o count --
↪barcodes align/Solo.out/Gene/filtered/barcodes.tsv --conversion TC align/Aligned.
↪sortedByCoord.out.bam
```

where `count` is the directory to which to save RNA quantifications. We provide a filtered barcode list `align/Solo.out/Gene/filtered/barcodes.tsv`, which was generated from the previous step, so that only these barcodes are processed during quantification. We specify the experimentally induced conversion with `--conversion`. In this example, our experiment introduces T-to-C conversions.

Once the above command finishes, the `count` directory will contain an `adata.h5ad` AnnData file containing all quantification results.

1.3.5 [Optional] Estimate

Optionally, we can estimate the unlabeled and labeled counts by statistically modelling the labeling dynamics (see *Statistical estimation*).

```
dynast estimate -o estimate count
```

where `estimate` is the directory to which to save RNA quantifications. We provide the directory that contains the quantification results (i.e. `-o` option of `dynast count`).

Once the above command finishes, the `estimate` directory will contain an `adata.h5ad` AnnData file containing all quantification and estimation results.

CHAPTER

TWO

PIPELINE USAGE

This sections covers basic usage of dynast.

2.1 Building the STAR index with `ref`

Internally, dynast uses the STAR RNA-seq aligner to align reads to the genome [Dobin2013]. Therefore, we must construct a STAR index to use for alignment. The `dynast ref` command is a wrapper around the STAR's `--runMode genomeGenerate` command, while also providing useful default parameters to limit memory usage, similar to Cell Ranger. Existing STAR indices can be used interchangeably with ones generated through dynast. A genome FASTA and gene annotation GTF are required to build the STAR index.

```
usage: dynast ref [-h] [--tmp TMP] [--keep-tmp] [--verbose] [-t THREADS] -i INDEX [-m␣
↪MEMORY] fasta gtf

Build a STAR index from a reference

positional arguments:
  fasta       Genomic FASTA file
  gtf         Reference GTF file

optional arguments:
  -h, --help  Show this help message and exit
  --tmp TMP   Override default temporary directory
  --keep-tmp  Do not delete the tmp directory
  --verbose   Print debugging information
  -t THREADS  Number of threads to use (default: 8)
  -m MEMORY   Maximum memory used, in GB (default: 16)

required arguments:
  -i INDEX    Path to the directory where the STAR index will be generated
```

2.2 Aligning FASTQs with `align`

The `dynast align` command is a wrapper around STARsolo [Dobin2013]. Dynast automatically formats the arguments to STARsolo to ensure the resulting alignment BAM contains information necessary for downstream processing.

Additionally, `align` sets a more lenient alignment score cutoff by setting `--outFilterScoreMinOverLread 0.3 --outFilterMatchNminOverLread 0.3` because the reads are expected to have experimentally-induced conversions. The STARsolo defaults for both are `0.66`. The `--STAR-overrides` argument can be used to pass arguments directly to STAR.

`dynast align` outputs a different set of BAM tags in the alignment BAM depending on the type of sequencing technology specified. These are described in the following subsections.

```
usage: dynast align [-h] [--tmp TMP] [--keep-tmp] [--verbose] [-t THREADS] -i INDEX [-o␣
↪OUT] -x TECHNOLOGY
                    [--strand {forward,reverse,unstranded}] [-w WHITELIST] [--overwrite]
                    [--STAR-overrides ARGUMENTS]
                    fastqs [fastqs ...]

Align FASTQs

positional arguments:
  fastqs                FASTQ files. If `-x smartseq`, this is a single manifest CSV␣
↪file where the first
                        column contains cell IDs and the next two columns contain paths␣
↪to FASTQs (the third
                        column may contain a dash `-` for single-end reads).

optional arguments:
  -h, --help            Show this help message and exit
  --tmp TMP             Override default temporary directory
  --keep-tmp            Do not delete the tmp directory
  --verbose             Print debugging information
  -t THREADS            Number of threads to use (default: 8)
  --strand {forward,reverse,unstranded}
                        Read strandedness. (default: `forward`)
  -w WHITELIST          Path to file of whitelisted barcodes to correct to. If not␣
↪provided, all barcodes are
                        used.
  --overwrite           Overwrite existing alignment files
  --STAR-overrides ARGUMENTS
                        Arguments to pass directly to STAR.

required arguments:
  -i INDEX              Path to the directory where the STAR index is located
  -o OUT                Path to output directory (default: current directory)
  -x TECHNOLOGY         Single-cell technology used. `dynast --list` to view all␣
↪supported technologies
```

2.2.1 UMI-based technologies

For UMI-based technologies (such as Drop-seq, 10X Chromium, scNT-seq), the following BAM tags are written to the alignment BAM.

- `MD`
- `HI`, `AS` for alignment index and score
- `CR`, `CB` for raw and corrected barcodes
- `UR`, `UB` for raw and corrected UMIs

2.2.2 Plate-based technologies

For plate-based technologies (such as Smart-Seq), the following BAM tags are written to the alignment BAM. See

- `MD`
- `HI`, `AS` for alignment index and score
- `RG` indicating the sample name

2.3 Calling consensus sequences with `consensus`

`dynast consensus` parses the alignment BAM to generate consensus sequences for each sequenced mRNA molecule (see *Consensus procedure*).

```
usage: dynast consensus [-h] [--tmp TMP] [--keep-tmp] [--verbose] [-t THREADS] -g GTF [-
↪o OUT] [--umi-tag TAG]
                        [--barcode-tag TAG] [--gene-tag TAG] [--strand {forward,reverse,
↪unstranded}]
                        [--quality QUALITY] [--barcodes TXT] [--add-RS-RI]
                        bam

Generate consensus sequences

positional arguments:
  bam                   Alignment BAM file that contains the appropriate UMI and barcode␣
↪tags, specifiable with
                        `--umi-tag`, and `--barcode-tag`.

optional arguments:
  -h, --help            Show this help message and exit
  --tmp TMP             Override default temporary directory
  --keep-tmp            Do not delete the tmp directory
  --verbose             Print debugging information
  -t THREADS            Number of threads to use (default: 8)
  -o OUT                Path to output directory (default: current directory)
  --umi-tag TAG         BAM tag to use as unique molecular identifiers (UMI). If not␣
↪provided, all reads are assumed
                        to be unique. (default: None)
  --barcode-tag TAG     BAM tag to use as cell barcodes. If not provided, all reads are␣
↪assumed to be from a single
```

(continues on next page)

(continued from previous page)

```
                        cell. (default: None)
  --gene-tag TAG        BAM tag to use as gene assignments (default: GX)
  --strand {forward,reverse,unstranded}
                        Read strandedness. (default: `forward`)
  --quality QUALITY     Base quality threshold. When generating a consensus nucleotide␣
↪at a certain position, the base
                        with smallest error probability below this quality threshold is␣
↪chosen. If no base meets this
                        criteria, the reference base is chosen. (default: 27)
  --barcodes TXT        Textfile containing filtered cell barcodes. Only these barcodes␣
↪will be processed.
  --add-RS-RI           Add custom RS and RI tags to the output BAM, each of which␣
↪contain a semi-colon delimited list
                        of read names (RS) and alignment indices (RI) of the reads and␣
↪alignments from which the
                        consensus is derived. This option is useful for debugging.

required arguments:
  -g GTF                Path to GTF file used to generate the STAR index
```

The resulting BAM will contain a collection of consensus alignments and a subset of original alignments (for those alignments for which a consensus could not be determined). For the latter, they will contain the following modified BAM tags.

- `GX`, `GN` each containing the assigned gene ID and name. Note that these tags are used regardless of what was provided to `--gene-tag`. Since these are reads for which a consensus could not be determined, these tags will be identical to what was contained in the tag provided with `--gene-tag`.

For the consensus reads, their names will be seemingly random sequences of letters and numbers (in reality, these are SHA256 checksums of the grouped read names). They will also contain the following modified BAM tags.

- `AS` is now the *sum* of the alignment scores of the reads
- `HI`, the alignment index, is always 1

and the follwing additional BAM tags.

- `RN` indicating how many reads were used to generate the consensus
- `RS`, `RI` each containing a semicolon-delimited list of read names and their corresponding alignment indices (`HI` tag in the original BAM) that were used to generate the consensus (only added if `--add-RS-RI` is provided)
- `GX`, `GN` each containing the assigned gene ID and name. Note that these tags are used regardless of what was provided to `--gene-tag`.

2.4 Quantifying counts with `count`

`dynast count` parses the alignment BAM and quantifies the four RNA species (unlabeled unspliced, unlabeled spliced, labeled unspliced, labeled spliced) and outputs the results as a ready-to-use AnnData `H5AD` file. In order to properly quantify the above four species, the alignment BAM must contain specific BAM tags, depending on what sequencing technology was used. If `dynast align` was used to generate the alignment BAM, dynast automatically configures the appropriate BAM tags to be written.

```
usage: dynast count [-h] [--tmp TMP] [--keep-tmp] [--verbose] [-t THREADS] -g GTF --
↪conversion CONVERSION [-o OUT]
                [--umi-tag TAG] [--barcode-tag TAG] [--gene-tag TAG] [--strand {forward,
↪reverse,unstranded,auto}]
                [--quality QUALITY] [--snp-threshold THRESHOLD] [--snp-min-coverage␣
↪THRESHOLD] [--snp-csv CSV]
                [--barcodes TXT] [--gene-names] [--no-splicing | --exon-overlap {lenient,
↪strict}] [--control]
                [--dedup-mode {auto,conversion,exon}] [--overwrite]
                bam

Quantify unlabeled and labeled RNA

positional arguments:
  bam                   Alignment BAM file that contains the appropriate UMI and barcode␣
↪tags, specifiable with
                        `--umi-tag`, and `--barcode-tag`.

optional arguments:
  -h, --help            Show this help message and exit
  --tmp TMP             Override default temporary directory
  --keep-tmp            Do not delete the tmp directory
  --verbose             Print debugging information
  -t THREADS            Number of threads to use (default: 8)
  -o OUT                Path to output directory (default: current directory)
  --umi-tag TAG         BAM tag to use as unique molecular identifiers (UMI). If not␣
↪provided, all reads are assumed
                        to be unique. (default: None)
  --barcode-tag TAG     BAM tag to use as cell barcodes. If not provided, all reads are␣
↪assumed to be from a single
                        cell. (default: None)
  --gene-tag TAG        BAM tag to use as gene assignments (default: GX)
  --strand {forward,reverse,unstranded,auto}
                        Read strandedness. By default, this is auto-detected from the␣
↪BAM.
  --quality QUALITY     Base quality threshold. Only bases with PHRED quality greater␣
↪than this value will be
                        considered when counting conversions. (default: 27)
  --snp-threshold THRESHOLD
                        Conversions with (# conversions) / (# reads) greater than this␣
↪threshold will be considered a
                        SNP and ignored. (default: no SNP detection)
  --snp-min-coverage THRESHOLD
                        For a conversion to be considered as a SNP, there must be at␣
↪least this many reads mapping to
                        that region. (default: 1)
  --snp-csv CSV         CSV file of two columns: contig (i.e. chromosome) and genome␣
↪position of known SNPs
  --barcodes TXT        Textfile containing filtered cell barcodes. Only these barcodes␣
↪will be processed.
  --gene-names          Group counts by gene names instead of gene IDs when generating␣
↪the h5ad file.
  --no-splicing, --transcriptome-only
```

(continues on next page)

(continued from previous page)

```
                        Do not assign reads a splicing status (spliced, unspliced,␣
↪ambiguous) and ignore reads that
                        are not assigned to the transcriptome.
  --exon-overlap {lenient,strict}
                        Algorithm to use to detect spliced reads (that overlap exons).␣
↪May be `strict`, which assigns
                        reads as spliced if it only overlaps exons, or `lenient`, which␣
↪assigns reads as spliced if it
                        does not overlap with any introns of at least one transcript.␣
↪(default: strict)
  --control             Indicate this is a control sample, which is used to detect SNPs.
  --dedup-mode {auto,conversion,exon}
                        Deduplication mode for UMI-based technologies (required `--umi-
↪tag`). Available choices are:
                        `auto`, `conversion`, `exon`. When `conversion` is used, reads␣
↪that have at least one of the
                        provided conversions is prioritized. When `exon` is used, exonic␣
↪reads are prioritized. By
                        default (`auto`), the BAM is inspected to select the appropriate␣
↪mode.
  --overwrite           Overwrite existing files.

required arguments:
  -g GTF                Path to GTF file used to generate the STAR index
  --conversion CONVERSION
                        The type of conversion(s) introduced at a single timepoint.␣
↪Multiple conversions can be
                        specified with a comma-delimited list. For example, T>C and A>G␣
↪is TC,AG. This option can be
                        specified multiple times (i.e. dual labeling), for each labeling␣
↪timepoint.
```

2.4.1 Basic arguments

The `--barcode-tag` and `--umi-tag` arguments are used to specify what BAM tags should be used to differentiate cells (barcode) and RNA molecules (UMI). If the former is not specified, all BAM alignments are assumed to be from a single cell, and if the latter is not specified, all aligned reads are assumed to be unique (i.e. no read deduplication is performed). If `align` was used to generate the alignment BAM, then `--barcode-tag CB --umi-tag UB` is recommended for UMI-based technologies (see *UMI-based technologies*), and `--barcode-tag RG` is recommended for Plate-based technologies (see *Plate-based technologies*).

The `--strand` argument can be used to specify the read strand of the sequencing technology. Usually, the default (`forward`) is appropriate, but this argument may be of use for other technologies.

The `--conversion` argument is used to specify the type of conversion that is experimentally introduced as a two-character string. For instance, a T>C conversion is represented as `TC`, which is the default. Multiple conversions can be specified as a comma-delimited list, and `--conversion` may be specified multiple times to indicate multiple-indexing experiments. For example, for an experiment that introduced T>C mutations at timepoint 1 and A>G and C>G mutations at timepoint 2, the appropriate options would be `--conversion TC --conversion AG,CG`.

The `--gene-names` argument can be used to specify that the resulting AnnData should contain gene names as its columns, instead of the usual gene IDs.

2.4.2 Detecting and filtering SNPs

`dynast count` has the ability to detect single-nucleotide polymorphisms (SNPs) by calculating the fraction of reads with a mutation at a certain genomic position. `--snp-threshold` can be used to specify the proportion threshold greater than which a SNP will be called at that position. All conversions/mutations at the genomic positions with SNPs detected in this manner will be filtered out from further processing. In addition, a CSV file containing known SNP positions can be provided with the `--snp-csv` argument. This argument accepts a CSV file containing two columns: contig (i.e. chromosome) and genomic position of known SNPs.

2.4.3 Read deduplication modes

The `--dedup-mode` option is used to select how duplicate reads should be deduplicated for UMI-based technologies (i.e. `--umi-tag` is provided). Two different modes are supported: `conversion` and `exon`. The former prioritizes reads that have at least one conversions provided by `--conversion`. The latter prioritizes exonic reads. See *quant* for a more technical description of how deduplication is performed. Additionally, see *Consensus procedure* to get an idea of why selecting the correct option may be important.

By default, the `--dedup-mode` is set to `auto`, which sets the deduplication mode to `exon` if the input BAM is detected to be a consensus-called BAM (a BAM generated with `dynast consensus`). Otherwise, it is set to `conversion`. This option has no effect for non-UMI technologies.

2.5 Estimating counts with `estimate`

The fraction of labeled RNA is estimated with the `dynast estimate` command. Whereas `dynast count` produces naive UMI count matrices, `dynast estimate` statistically models labeling dynamics to estimate the true fraction of labeled RNA (and then in turn uses this fraction to split the total UMI counts into unlabeled and labeled RNA). See *Statistical estimation* of a technical overview of this process. In this section, we will simply be describing the command-line usage of this command.

```
usage: dynast estimate [-h] [--tmp TMP] [--keep-tmp] [--verbose] [-t THREADS] [--reads
↪{total,transcriptome,spliced,unspliced}] [-o OUT] [--barcodes TXT]
                       [--groups CSV] [--method {pi_g,alpha}] [--ignore-groups-for-est] [--
↪genes TXT] [--cell-threshold COUNT] [--cell-gene-threshold COUNT]
                       [--gene-names] [--downsample NUM] [--downsample-mode MODE] [--
↪control] [--p-e P_E]
                       count_dirs [count_dirs ...]

Estimate fraction of labeled RNA

positional arguments:
  count_dirs            Path to directory that contains `dynast count` output. When␣
↪multiple are provided, the barcodes in each of the count directories are
                        suffixed with `-i` where i is a 0-indexed integer.

optional arguments:
  -h, --help            Show this help message and exit
  --tmp TMP             Override default temporary directory
  --keep-tmp            Do not delete the tmp directory
  --verbose             Print debugging information
  -t THREADS            Number of threads to use (default: 8)
  --reads {total,transcriptome,spliced,unspliced}
```

(continues on next page)

(continued from previous page)

```
                        Read groups to perform estimation on. This option can be used␣
↪multiple times to estimate multiple groups. (default: all possible reads
                        groups)
  -o OUT                Path to output directory (default: current directory)
  --barcodes TXT        Textfile containing filtered cell barcodes. Only these barcodes␣
↪will be processed. This option may be used multiple times when
                        multiple input directories are provided.
  --groups CSV          CSV containing cell (barcode) groups, where the first column is␣
↪the barcode and the second is the group name the cell belongs to.
                        Estimation will be performed by aggregating UMIs per group. This␣
↪option may be used multiple times when multiple input directories are
                        provided.
  --method {pi_g,alpha}
                        Correction method to use. May be `pi_g` to estimate the fraction␣
↪of labeled RNA for every cell-gene combination, or `alpha` to use
                        alpha correction as used in the scNT-seq paper. `alpha` is␣
↪recommended for UMI-based assays. This option has no effect when used with
                        `--control`. (default: alpha)
  --ignore-groups-for-est
                        Ignore cell groupings when calculating final estimations for the␣
↪fraction of labeled RNA. When `--method pi_g`, groups are ignored
                        when estimating fraction of labeled RNA. When `--method alpha`,␣
↪groups are ignored when estimating detection rate. This option only
                        has an effect when `--groups` is also specified.
  --genes TXT           Textfile containing list of genes to use. All other genes will␣
↪be treated as if they do not exist.
  --cell-threshold COUNT
                        A cell must have at least this many reads for correction.␣
↪(default: 1000)
  --cell-gene-threshold COUNT
                        A cell-gene pair must have at least this many reads for␣
↪correction. Only for `--method pi_g`. (default: 16)
  --gene-names          Group counts by gene names instead of gene IDs when generating␣
↪H5AD file
  --downsample NUM      Downsample the number of reads (UMIs). If a decimal between 0␣
↪and 1 is given, then the number is interpreted as the proportion of
                        remaining reads. If an integer is given, the number is␣
↪interpreted as the absolute number of remaining reads.
  --downsample-mode MODE
                        Downsampling mode. Can be one of: `uniform`, `cell`, `group`. If␣
↪`uniform`, all reads (UMIs) are downsampled uniformly at random. If
                        `cell`, only cells that have more reads than the argument to `--
↪downsample` are downsampled to exactly that number. If `group`,
                        identical to `cell` but per group specified by `--groups`.
  --control             Indicate this is a control sample, only the background mutation␣
↪rate will be estimated.
  --p-e P_E             Textfile containing a single number, indicating the estimated␣
↪background mutation rate
```

2.5.1 Estimation methods

Dynast supports two different statistical correction methods. The `--method pi_g` employs a Bayesian inference approach to directly estimate the fraction of labeled RNA for each cell-gene combination. While this approach performs well for plate-based assays (such as those using Smart-Seq), droplet-based assays (such as those using Drop-seq) produce very sparse counts for which this estimation procedure often fails due to low number of reads per cell-gene. Therefore, `--method alpha` uses the detection rate estimation used in [Qiu2020], which is more suited for sparse data. See *Bayesian inference (\pi_g)* for more information.

2.5.2 Estimation thresholds

The `--cell-threshold` and `--cell-gene-threshold` arguments control the minimum number of reads that a cell and cell-gene combination must have for accurate estimation. By default, these are `1000` and `16` respectively. Any cells with reads less than the former are excluded from estimation, and the same goes for any genes within a cell that has less reads than the latter. If `--groups` is also provided, then these thresholds apply to each cell **group** instead of each cell individually. Internally, `--cell-threshold` is used to filter cells before estimating the average conversion rate in labeled RNA (see *Induced rate estimation (p_c)*), and `--cell-gene-threshold` is used to filter cell-gene combinations before estimating the fraction of new RNA and only has an effect when `--method pi_g` (see *Bayesian inference (\pi_g)*).

2.5.3 Estimation on a subset of RNA species

The `--reads` argument controls which RNA species to run the estimation procedure on. By default, all possible RNA species, minus `ambiguous` reads, are used. This argument can take on the following values: `total`, `transcriptome`, `spliced`, `unspliced` (see *Read groups*). The value of this argument specifies which group of unlabeled/labeled RNA counts will be estimated. For instance, `--reads spliced` will run statistical estimation on unlabeled/labeled spliced reads. This option may be provided multiple times to run estimation on multiple groups. The procedure involves estimating the conversion rate of unlabeled and labeled RNA, and modeling the fraction of new RNA as a binomial mixture model (see *Statistical estimation*).

2.5.4 Grouping cells

Sometimes, grouping read counts across cells may provide better estimation results, especially in the case of droplet-based methods, which result in fewer reads per cell and gene compared to plate-based methods. The `--groups` argument can be used to provide a CSV of two columns: the first containing the cell barcodes and the second containing group names that each cell belongs to. Estimation is then performed on a per-group basis by combining the read counts across all cells in each group. This strategy may be applied across different samples, simply by specifying multiple input directories. In this case, the number of group CSVs specified with `--groups` must match the number of input directories. For example, when providing two input directories `./input1` and `./input2`, with the intention of grouping cells across these two samples, two group CSVs, `groups1.csv` and `groups2.csv` must be provided where the former are groups for barcodes in the first sample, and the latter are groups for barcodes in the second sample. The group names may be shared across samples. The output AnnData will still contain reads per cell.

Cell groupings provided this way may be ignored for estimation of the fraction of labeled RNA when `--method pi_g` or the detection rate when `--method alpha` (see *Bayesian inference (\pi_g)*) by providing the `--ignore-groups-for-est` flag. This flag may be used only in conjunction with `--groups`, and when it is provided, final estimation is performed per cell, while estimation of background and induced mutation rates are still done per group.

2.5.5 Downsampling

Downsampling UMIs uniformly, per cell, or per cell group may be useful to significantly reduce runtime while troubleshooting pipeline parameters (or just to quickly get some preliminary results). Dynast can perform downsampling when the `--downsample` argument is used. The value of this argument may either be an integer indicating the number of UMIs to retain or a proportion between 0 and 1 indicating the proportion of UMIs to retain. Additionally, the downsampling mode may be specified with the `--downsample-mode` argument, which takes one of the following three parameters: `uniform`, `cell`, `group`. `uniform` is the default that downsamples UMIs uniformly at random. When `cell` is provided, the value of `--downsample` may only be an integer specifying the threshold to downsample cells to. Only cells with UMI counts greater than this value will be downsampled to exactly this value. `group` works the same way, but for cell groups and may be used only in conjunction with `--groups`.

2.6 Control samples

Control samples may be used to find common SNPs and directly estimate the conversion rate of unlabeled RNA (see *Background estimation (p_e)*). Normally, the latter is estimating using the reads directly. However, it is possible to use a control sample (prepared in absence of the experimental introduction of conversions) to calculate this value directly. In addition, SNPs can be called in the control sample, and these called SNPs can be used when running the test sample(s) (see *Detecting and filtering SNPs* for SNP arguments). Note that SNP calling is done with `dynast count`.

A typical workflow for a control sample is the following.

```
dynast count --control --snp-threshold 0.5 [...] -o control_count --conversion TC -g GTF.
↪gtf CONTROL.bam
dynast estimate --control -o control_estimate control_count
```

Where `[...]` indicates the usual options that would be used for `dynast count` if this were not control samples. See *Basic arguments* for these options.

The `dynast count` command detects SNPs from the control sample and outputs them to the file `snps.csv` in the output directory `control_count`. The `dynast estimate` calculates the background conversion rate of unlabeled RNA to the file `p_e.csv` in the output directory `control_estimate`. These files can then be used as input when running the test sample.

```
dynast count --snp-csv control_count/snps.csv -o test_count [...] INPUT.bam
dynast estimate --p-e control_estimate/p_e.csv -o test_estimate test_count
```

The above set of commands runs quantification and estimation on the test sample using the SNPs detected from the control sample (`control_count/snps.csv`) and the background conversion rate estimated from the control sample (`control_estimate/p_e.csv`).

CHAPTER

THREE

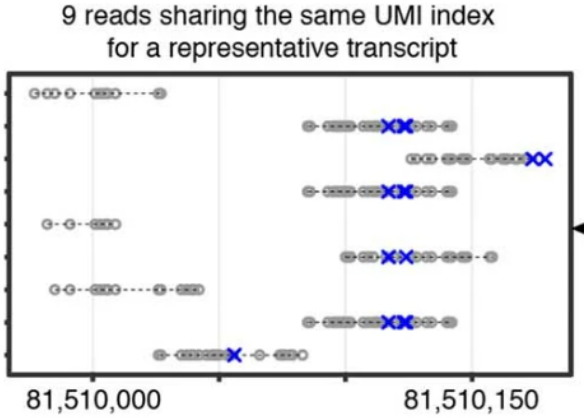
TECHNICAL INFORMATION

This section details technical information of the quantification and statistical estimation procedures of the `dynast consensus`, `dynast count` and `dynast estimate` commands. Descriptions of `dynast ref` and `dynast align` commands are in *Pipeline Usage*.

3.1 Consensus procedure

`dynast consensus` procedure generates consensus sequences for each mRNA molecule that was present in the sample. It relies on sequencing the same mRNA molecule (often distinguished using UMIs for UMI-based technologies, or start and end alignment positions for non-UMI technologies) multiple times, to obtain a more confident read sequence.

Why don't we just perform UMI-deduplication (by just selecting a single read among the reads that share the same UMI) and call it a day? Though it seems counterintuitive, reads sharing the same UMI may originate from different regions of the same mRNA, as [Qiu2020] (scNT-seq) observed in Extended Data Fig.1b.



Therefore, simply selecting one read and discarding the rest will cause a bias toward unlabeled reads because the selected read may happen to have no conversions, while all the other (discarded) reads do. Therefore, we found it necessary to implement a consensus-calling procedure, which works in the following steps. Here, we assume cell barcodes are available (`--barcode-tag` is provided), but the same procedure can be performed in the absence of cell barcodes by assuming all reads were from a single cell. Additionally, we will use the term *read* and *alignment* interchangeably because only a single alignment (see the note below) from each read will be considered.

1. Alignments in the BAM are grouped into UMI groups. In the case of UMI-based technologies (`--umi-tag` is provided), a UMI group is defined as the set of alignments that share the same cell barcode, UMI, and gene. For alignments with the `--gene-tag` tag, assigning these into a UMI group is trivial. For alignments without this tag, it is assigned to the gene whose gene body fully contains the alignment. If multiple genes are applicable, the

alignment is not assigned a UMI group and output to the resulting BAM as-is. For non-UMI-based technologies, the start and end positions of the alignment are used as a pseudo-UMI.

2. For each UMI group, the consensus base is taken for every genomic location that is covered by at least one alignment in the group. The consensus base is defined as the base with the highest sum of quality scores of that base among all alignments in the group. Loosely, this is proportional to the conditional probability of each base being a sequencing error. If the consensus base score does not exceed the score specified with `--quality`, then the reference base is taken instead. Once this is done for every covered genomic location, the consensus alignment is output to the BAM, and the UMI group is discarded (i.e. not written to the BAM).

Note: Only primary, not-duplicate, mapped BAM entries are considered (equivalent to the `0x4`, `0x100`, `0x400` BAM flags being unset). For paired reads, only properly paired alignments (`0x2` BAM flag being set) are considered. Additionally, if `--barcode-tag` or `--umi-tag` are provided, only BAM entries that have these tags are considered. Any alignments that do not satisfy all of these conditions are not written to the output BAM.

3.1.1 Consensus rules

Here are the rules that are used to select a consensus nucleotide for a certain genomic positions, assuming `--quality 27`.

1. Nucleotide with highest sum of base quality scores, given that the sum is greater than or equal to `--quality`.

Source	Nucleotide	Quality score
Reference	A	-
Read 1	C	10
Read 2	C	20
Read 3	A	20
Consensus	**C**	**30**

2. Reference nucleotide, if all of the sum of quality scores are less than `--quality`.

Source	Nucleotide	Quality score
Reference	A	-
Read 1	C	10
Read 2	C	10
Consensus	**A**	**20**

3. When there is a tie in the highest sum of base quality scores and the reference nucleotide is one of them, the reference nucleotide.

Source	Nucleotide	Quality score
Reference	A	-
Read 1	C	30
Read 2	A	30
Consensus	**A**	**30**

4. When there is a tie in the highest sum of base quality scores and the reference nucleotide is not one of them, the first nucleotide in lexicographic order (A, C, G, T).

Source	Nucleotide	Quality score
Reference	T	-
Read 1	C	30
Read 2	A	30
Consensus	**A**	**30**

3.2 Count procedure

`dynast count` procedure consists of three steps:

1. *parse*
2. *snp*
3. *quant*

3.2.1 `parse`

1. All gene and transcript information are parsed from the gene annotation GTF (`-g`) and saved as Python pickles `genes.pkl.gz` and `transcripts.pkl.gz`, respectively.
2. All aligned reads are parsed from the input BAM and output to `conversions.csv` and `alignments.csv`. The former contains a line for every conversion, and the latter contains a line for every alignment. Note that no conversion filtering (`--quality`) is performed in this step. Two `.idx` files are also output, corresponding to each of these CSVs, which are used downstream for fast parsing. Splicing types are also assigned in this step if `--no-splicing` was not provided.

Note: Only primary, not-duplicate, mapped BAM entries are considered (equivalent to the `0x4`, `0x100`, `0x400` BAM flags being unset). For paired reads, only properly paired alignments (`0x2` BAM flag being set) are considered. Additionally, if `--barcode-tag` or `--umi-tag` are provided, only BAM entries that have these tags are considered.

3.2.2 `snp`

This step is skipped if `--snp-threshold` is not specified.

1. Read coverage of the genome is computed by parsing all aligned reads from the input BAM and output to `coverage.csv`.
2. SNPs are detected by calculating, for every genomic position, the fraction of reads with a conversion at that position over its coverage. If this fraction is greater than `--snp-threshold`, then the genomic position and the specific conversion is written to the output file `snps.csv`. Any conversion with PHRED quality less than or equal to `--quality` is not counted as a conversion. Additionally, `--snp-min-coverage` can be used to specify the minimum coverage any detected SNP must have. Any sites that have less than this coverage are ignored (and therefore not labeled as SNPs).

3.2.3 quant

1. For every read, the numbers of each conversion (A>C, A>G, A>T, C>A, etc.) and nucleotide content (how many of A, C, G, T there are in the region that the read aligned to) are counted. Any SNPs provided with `--snp-csv` or detected from the *snp* step are not counted. If both are present, the union is used. Additionally, Any conversion with PHRED quality less than or equal to `--quality` is not counted as a conversion.
2. For UMI-based technologies, reads are deduplicated by the following order of priority: 1) reads that have at least one conversion specified with `--conversion`, 2) read that aligns to the transcriptome (i.e. exon-only), 3) read that has the highest alignment score, and 4) read with the most conversions specified with `--conversion`. If multiple conversions are provided, the sum is used. Reads are considered duplicates if they share the same barcode, UMI, and gene assignment. For plate-based technologies, read deduplication should have been performed in the alignment step (in the case of STAR, with the `--soloUMIdedup Exact`), but in the case of multimapping reads, it becomes a bit more tricky. If a read is multimapping such that some alignments map to the transcriptome while some do not, the transcriptome alignment is taken (there can not be multiple transcriptome alignments, as this is a constraint within STAR). If none align to the transcriptome and the alignments are assigned to multiple genes, the read is dropped, as it is impossible to assign the read with confidence. If none align to the transcriptome and the alignments are assigned multiple velocity types, the velocity type is manually set to `ambiguous` and the first alignment is kept. If none of these cases are true, the first alignment is kept. The final deduplicated/de-multimapped counts are output to `counts_{conversions}.csv`, where `{conversions}` is an underscore-delimited list of all conversions provided with `--conversion`.

Note: All bases in this file are relative to the forward genomic strand. For example, a read mapped to a gene on the reverse genomic strand should be complemented to get the actual bases.

3.2.4 Output Anndata

All results are compiled into a single AnnData `H5AD` file. The AnnData object contains the following:

- The *transcriptome* read counts in `.X`. Here, *transcriptome* reads are the mRNA read counts that are usually output from conventional scRNA-seq quantification pipelines. In technical terms, these are reads that contain the BAM tag provided with the `--gene-tag` (default is `GX`).
- Unlabeled and labeled *transcriptome* read counts in `.layers['X_n_{conversion}']` and `.layers['X_l_{conversion}']`.

The following layers are also present if `--no-splicing` or `--transcriptome-only` was *NOT* specified.

- The *total* read counts in `.layers['total']`.
- Unlabeled and labeled *total* read counts in `.layers['unlabeled_{conversion}']` and `.layers['labeled_{conversion}']`.
- Spliced, unspliced and ambiguous read counts in `.layers['spliced']`, `.layers['unspliced']` and `.layers['ambiguous']`.
- Unspliced unlabeled, unspliced labeled, spliced unlabeled, spliced labeled read counts in `.layers['un_{conversion}']`, `.layers['ul_{conversion}']`, `.layers['sn_{conversion}']` and `.layers['sl_{conversion}']` respectively.

The following equalities always hold for the resulting Anndata.

- `.layers['total'] == .layers['spliced'] + .layers['unspliced'] + .layers['ambiguous']`

The following additional equalities always hold for the resulting Anndata in the case of single labeling (`--conversion` was specified once).

- `.X == .layers['X_n_{conversion}'] + .layers['X_l_{conversion}']`
- `.layers['spliced'] == .layers['sn_{conversion}'] + .layers['sl_{conversion}']`
- `.layers['unspliced'] == .layers['un_{conversion}'] + .layers['ul_{conversion}']`

Tip: To quantify splicing data from conventional scRNA-seq experiments (experiments without metabolic labeling), we recommend using the kallisto | bustools pipeline.

3.3 Estimate procedure

`dynast estimate` procedure consists of two steps:

1. *aggregate*
2. *estimate*

3.3.1 `aggregate`

For each cell and gene and for each conversion provided with `--conversion`, the conversion counts are aggregated into a CSV file such that each row contains the following columns: cell barcode, gene, conversion count, nucleotide content of the original base (i.e. if the conversion is T>C, this would be T), and the number of reads that have this particular barcode-gene-conversion-content combination. This procedure is done for all read groups that exist (see *Read groups*).

3.3.2 `estimate`

1. The background conversion rate p_e is estimated, if `--p-e` was not provided (see *Background estimation (p_e)*). If `--p-e` was provided, this value is used and estimation is skipped. p_e.
2. The induced conversion rate p_c is estimated using an expectation maximization (EM) approach, for each conversion provided with `--conversion` (see *Induced rate estimation (p_c)*). p_c where `{conversion}` is an underscore-delimited list of each conversion (because multiple conversions can be introduced in a single timepoint). This step is skipped for control samples with `--control`.
3. Finally, the counts are split into estimated number of labeled and unlabeled counts. These may be produced by either estimating the the fraction of labeled RNA per cell-gene π_g directly by using `--method pi_g` or using a detection rate estimation-based method by using `--method alpha` (see *Bayesian inference (\pi_g)*). By default, the latter is performed. The resulting estimated fractions are written to CSV files named `pi_xxx.csv`, where the former contains estimations per cell-gene (`--method pi_g`) or per cell (`--method alpha`).

Note: The induced conversion rate p_c estimation always uses all reads present in the counts CSV (located within the count directory provided to the `dynast estimate` command). Therefore, unless `--no-splicing` or `--transcriptome-only` was provided to `dynast count`, *total* reads will be used.

3.3.3 Output Anndata

All results are compiled into a single AnnData `H5AD` file. The AnnData object contains the following:

- The *transcriptome* read counts in `.X`. Here, *transcriptome* reads are the mRNA read counts that are usually output from conventional scRNA-seq quantification pipelines. In technical terms, these are reads that contain the BAM tag provided with the `--gene-tag` (default is `GX`).
- Unlabeled and labeled *transcriptome* read counts in `.layers['X_n_{conversion}']` and `.layers['X_l_{conversion}']`. If `--reads transcriptome` was specified, the estimated counts are in `.layers['X_n_{conversion}_est']` and `.layers['X_l_{conversion}_est']`. `{conversion}` is an underscore-delimited list of each conversion provided with `--conversion` when running `dynast count`.
- If `--method pi_g`, the estimated fraction of labeled RNA per cell-gene in `.layers['{group}_{conversion}_pi_g']`.
- If `--method alpha`, the per cell detection rate in `.obs['{group}_{conversion}_alpha']`.

The following layers are also present if `--no-splicing` or `--transcriptome-only` was *NOT* specified when running `dynast count`.

- The *total* read counts in `.layers['total']`.
- Unlabeled and labeled *total* read counts in `.layers['unlabeled_{conversion}']` and `.layers['labeled_{conversion}']`. If `--reads total` is specified, the estimated counts are in `.layers['unlabeled_{conversion}_est']` and `.layers['labeled_{conversion}_est']`.
- Spliced, unspliced and ambiguous read counts in `.layers['spliced']`, `.layers['unspliced']` and `.layers['ambiguous']`.
- Unspliced unlabeled, unspliced labeled, spliced unlabeled, spliced labeled read counts in `.layers['un_{conversion}']`, `.layers['ul_{conversion}']`, `.layers['sn_{conversion}']` and `.layers['sl_{conversion}']` respectively. If `--reads spliced` and/or `--reads unspliced` was specified, layers with estimated counts are added. These layers are suffixed with `_est`, analogous to *total* counts above.

In addition to the equalities listed in the *quant* section, the following inequalities always hold for the resulting Anndata.

- `.X >= .layers['X_n_{conversion}_est'] + .layers['X_l_{conversion}_est']`
- `.layers['spliced'] >= .layers['sn_{conversion}_est'] + .layers['sl_{conversion}_est']`
- `.layers['unspliced'] >= .layers['un_{conversion}_est'] + .layers['ul_{conversion}_est']`

Tip: To quantify splicing data from conventional scRNA-seq experiments (experiments without metabolic labeling), we recommend using the kallisto | bustools pipeline.

3.3.4 Caveats

The statistical estimation procedure described above comes with some caveats.

- The induced conversion rate (p_c) can not be estimated for cells with too few reads (defined by the option `--cell-threshold`).
- The fraction of labeled RNA (π_g) can not be estimated for cell-gene combinations with too few reads (defined by the option `--cell-gene-threshold`).

For statistical definitions of these variables, see *Statistical estimation*.

Therefore, for low coverage data, we expect many cell-gene combinations to not have any estimations in the Anndata layers prefixed with `_est`, indicated with zeros. It is possible to construct a boolean mask that contains `True` for cell-gene combinations that were successfully estimated and `False` otherwise. Note that we are using *total* reads.

```
estimated_mask = ((adata.layers['labeled_{conversion}'] + adata.layers['unlabeled_
↪{conversion}']) > 0) & \
    ((adata.layers['labeled_{conversion}_est'] + adata.layers['unlabeled_{conversion}_est
↪']) > 0)
```

Similarly, it is possible to construct a boolean mask that contains `True` for cell-gene combinations for which estimation failed (either due to having too few reads mapping at the cell level or the cell-gene level) and `False` otherwise.

```
failed_mask = ((adata.layers['labeled_{conversion}'] + adata.layers['unlabeled_
↪{conversion}']) > 0) & \
    ((adata.layers['labeled_{conversion}_est'] + adata.layers['unlabeled_{conversion}_est
↪']) == 0)
```

The same can be done with other *Read groups*.

3.4 Read groups

Dynast separates reads into read groups, and each of these groups are processed together.

- `total`: All reads. Used only when `--no-splicing` or `--transcriptome-only` is not used.
- `transcriptome`: Reads that map to the transcriptome. These are reads that have the `GX` tag in the BAM (or whatever you provide for the `--gene-tag` argument). This group also represents all reads when `--no-splicing` or `--transcriptome-only` is used.
- `spliced`: Spliced reads
- `unspliced`: Unspliced reads
- `ambiguous`: Ambiguous reads

The latter three groups are mutually exclusive.

3.5 Statistical estimation

Dynast can statistically estimate unlabeled and labeled RNA counts by modeling the distribution as a binomial mixture model [Jürges2018]. Statistical estimation can be run with `dynast estimate` (see *estimate*).

3.5.1 Overview

First, we define the following model parameters. For the remainder of this section, let the conversion be T>C. Note that all parameters are calculated per barcode (i.e. cell) unless otherwise specified.

p_e : average conversion rate in unlabeled RNA

p_c : average conversion rate in labeled RNA

π_g : fraction of labeled RNA for gene g

y : number of observed T>C conversions (in a read)

n : number of T bases in the genomic region (a read maps to)

Then, the probability of observing k conversions given the above parameters is

$$\mathbb{P}(k; p_e, p_c, n, \pi) = (1 - \pi_g)B(k; n, p_e) + \pi_g B(k; n, p_c)$$

where $B(k, n, p)$ is the binomial PMF. The goal is to calculate π_g, which can be used the split the raw counts to get the estimated counts. We can extract k and n directly from the read alignments, while calculating p_e and p_c is more complicated (detailed below).

3.5.2 Background estimation (p_e)

If we have control samples (i.e. samples without the conversion-introducing treatment), we can calculate p_e directly by simply calculating the mutation rate of T to C. This is exactly what dynast does for `--control` samples. All cells are aggregated when calculating p_e for control samples.

Otherwise, we need to use other mutation rates as a proxy for the real T>C background mutation rate. In this case, p_e is calculated as the average conversion rate of all non-T bases to any other base. Mathematically,

$$p_e = average(r(A, C), r(A, G), \cdots, r(G, T))$$

where $r(X, Y)$ is the observed conversion rate from X to Y, and $average$ is the function that calculates the average of its arguments. Note that we do not use the conversion rates of conversions that start with a T. This is because T>C is our induced mutation, and this artificially deflates the T>A, T>G mutation rates (which can skew our p_e estimation to be lower than it should). In the event that multiple conversions are of interest, and they span all four bases as the initial base, then p_e estimation falls back to using all other conversions (regardless of start base).

3.5.3 Induced rate estimation (p_c)

p_c is estimated via an expectation maximization (EM) algorithm by constructing a sparse matrix A where each element $a_{k,n}$ is the number of reads with k T>C conversions and n T bases in the genomic region that each read align to. Assuming $p_e < p_c$, we treat $a_{k,n}$ as missing data if greater than or equal to 1% of the count is expected to originate from the p_e component. Mathematically, $a_{k,n}$ is excluded if

$$e_{k,n} = B(k, n, p_e) \cdot \sum_{k' \geq k} a_{k',n} > 0.01 a_{k,n}$$

Let $X = \{(k_1, n_1), \cdots\}$ be the excluded data. The E step fills in the excluded data by their expected values given the current estimate $p_c^{(t)}$,

$$a_{k,n}^{(t+1)} = \frac{\sum_{(k',n) \notin X} B(k, n, p_c^{(t)}) \cdot a_{k',n}}{\sum_{(k',n) \notin X} B(k', n, p_c^{(t)})}$$

The M step updates the estimate for p_c

$$p_c^{(t+1)} = \frac{\sum_{k,n} k a_{k,n}^{(t+1)}}{\sum_{k,n} n a_{k,n}^{(t+1)}}$$

3.5.4 Bayesian inference (π_g)

The fraction of labeled RNA is estimated with Bayesian inference using the binomial mixture model described above. A Markov chain Monte Carlo (MCMC) approach is applied using the p_e, p_c, and the matrix A found/estimated in previous steps. This estimation procedure is implemented with pyStan, which is a Python interface to the Bayesian inference package Stan. The Stan model definition is here.

When `--method pi_g`, this estimation yields the fraction of labeled RNA per cell-gene, π_g, which can be used directly to split the total RNA. However, when `--method alpha`, this estimation yields the fraction of labeled RNA per cell, π_c. As was described in [Qiu2020], the detection rate per cell, α_c, is calculated as

$$\alpha_c = \frac{\pi_c}{L_c + U_c}$$

where L_c and U_c are the numbers of labeled and unlableed RNA for cell c. Then, using this detection rate, the corrected labeled RNA is calculated as

$$N_c' = \min\left(\frac{L_c}{\alpha_c}, L_c + U_c\right)$$

CHAPTER

FOUR

DYNAST

4.1 Subpackages

4.1.1 dynast.benchmarking

Submodules

dynast.benchmarking.simulation

Module Contents

Functions

generate_sequence(k, seed=None)	
simulate_reads(sequence, p_e, p_c, pi, l=100, n=100, seed=None)	
initializer(model)	
estimate(df_counts, p_e, p_c, pi, estimate_p_e=False, estimate_p_c=False, estimate_pi=True, model=None, nasc=False)	
_simulate(p_e, p_c, pi, sequence=None, k=10000, l=100, n=100, estimate_p_e=False, estimate_p_c=False, estimate_pi=True, seed=None, model=None, nasc=False)	
simulate(p_e, p_c, pi, sequence=None, k=10000, l=100, n=100, n_runs=16, n_threads=8, estimate_p_e=False, estimate_p_c=False, estimate_pi=True, model=None, nasc=False)	
simulate_batch(p_e, p_c, pi, l, n, estimate_p_e, estimate_p_c, estimate_pi, n_runs, n_threads, model, nasc=False)	Helper function to run simulations in batches.
plot_estimations(X, Y, n_runs, means, truth, ax=None, box=True, tick_decimals=1, title=None, xlabel=None, ylabel=None)	

Attributes

__model

_pi_model

dynast.benchmarking.simulation.**generate_sequence**(*k*, *seed=None*)

dynast.benchmarking.simulation.**simulate_reads**(*sequence*, *p_e*, *p_c*, *pi*, *l=100*, *n=100*, *seed=None*)

dynast.benchmarking.simulation.**__model**

dynast.benchmarking.simulation.**_pi_model**

dynast.benchmarking.simulation.**initializer**(*model*)

dynast.benchmarking.simulation.**estimate**(*df_counts*, *p_e*, *p_c*, *pi*, *estimate_p_e=False*, *estimate_p_c=False*, *estimate_pi=True*, *model=None*, *nasc=False*)

dynast.benchmarking.simulation.**_simulate**(*p_e*, *p_c*, *pi*, *sequence=None*, *k=10000*, *l=100*, *n=100*, *estimate_p_e=False*, *estimate_p_c=False*, *estimate_pi=True*, *seed=None*, *model=None*, *nasc=False*)

dynast.benchmarking.simulation.**simulate**(*p_e*, *p_c*, *pi*, *sequence=None*, *k=10000*, *l=100*, *n=100*, *n_runs=16*, *n_threads=8*, *estimate_p_e=False*, *estimate_p_c=False*, *estimate_pi=True*, *model=None*, *nasc=False*)

dynast.benchmarking.simulation.**simulate_batch**(*p_e*, *p_c*, *pi*, *l*, *n*, *estimate_p_e*, *estimate_p_c*, *estimate_pi*, *n_runs*, *n_threads*, *model*, *nasc=False*)

Helper function to run simulations in batches.

dynast.benchmarking.simulation.**plot_estimations**(*X*, *Y*, *n_runs*, *means*, *truth*, *ax=None*, *box=True*, *tick_decimals=1*, *title=None*, *xlabel=None*, *ylabel=None*)

4.1.2 dynast.estimation

Submodules

dynast.estimation.alpha

Module Contents

Functions

read_alpha(alpha_path: str, group_by: Optional[List[str]] = None) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]	Read alpha CSV as a dictionary, with *group_by* columns as keys.
estimate_alpha(df_counts: pandas.DataFrame, pi_c: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], alpha_path: str, conversions: FrozenSet[str] = frozenset({'TC'}), group_by: Optional[List[str]] = None, pi_c_group_by: Optional[List[str]] = None) → str	Estimate the detection rate alpha.

dynast.estimation.alpha.**read_alpha**(*alpha_path: str*, *group_by: Optional[List[str]] = None*) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]

Read alpha CSV as a dictionary, with *group_by* columns as keys.

Parameters

alpha_path
Path to CSV containing alpha values

group_by
Columns to group by, defaults to *None*

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.alpha.**estimate_alpha**(*df_counts: pandas.DataFrame*, *pi_c: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]*, *alpha_path: str*, *conversions: FrozenSet[str] = frozenset({'TC'})*, *group_by: Optional[List[str]] = None*, *pi_c_group_by: Optional[List[str]] = None*) → str

Estimate the detection rate alpha.

Parameters

df_counts
Pandas dataframe containing conversion counts

pi_c
Labeled mutation rate

alpha_path
Path to output CSV containing alpha estimates

conversions
Conversions to consider

group_by
Columns to group by

pi_c_group_by
Columns that were used to group when calculating pi_c

Returns
Path to output CSV containing alpha estimates

dynast.estimation.p_c

Module Contents

Functions

read_p_c(p_c_path: str, group_by: Optional[List[str]] = None) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]	Read p_c CSV as a dictionary, with *group_by* columns as keys.
binomial_pmf(k: int, n: int, p: int) → float	Numbaized binomial PMF function for faster calculation.
expectation_maximization_nasc(values: numpy.ndarray, p_e: float, threshold: float = 0.01) → float	NASC-seq pipeline variant of the EM algorithm to estimate average
expectation_maximization(values: numpy.ndarray, p_e: float, p_c: float = 0.1, threshold: float = 0.01, max_iters: int = 300) → float	Run EM algorithm to estimate average conversion rate in labeled RNA.
estimate_p_c(df_aggregates: pandas.DataFrame, p_e: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], p_c_path: str, group_by: Optional[List[str]] = None, threshold: int = 1000, n_threads: int = 8, nasc: bool = False) → str	Estimate the average conversion rate in labeled RNA.

dynast.estimation.p_c.**read_p_c**(*p_c_path: str*, *group_by: Optional[List[str]] = None*) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]

Read p_c CSV as a dictionary, with *group_by* columns as keys.

Parameters

p_c_path
Path to CSV containing p_c values

group_by
Columns to group by, defaults to *None*

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.p_c.**binomial_pmf**(*k: int*, *n: int*, *p: int*) → float

Numbaized binomial PMF function for faster calculation.

Parameters

k
Number of successes

n
Number of trials

p
Probability of success

Returns

Probability of observing *k* successes in *n* trials with probability
of success *p*

dynast.estimation.p_c.**expectation_maximization_nasc**(*values: numpy.ndarray*, *p_e: float*, *threshold: float = 0.01*) → float

NASC-seq pipeline variant of the EM algorithm to estimate average conversion rate in labeled RNA.

Parameters

values
N x C Numpy array where N is the number of conversions, C is the nucleotide content, and the value at this position is the number of reads observed

p_e
Background mutation rate of unlabeled RNA

threshold
Filter threshold

Returns
Estimated conversion rate

dynast.estimation.p_c.**expectation_maximization**(*values: numpy.ndarray*, *p_e: float*, *p_c: float = 0.1*, *threshold: float = 0.01*, *max_iters: int = 300*) → float

Run EM algorithm to estimate average conversion rate in labeled RNA.

This function runs the following two steps. 1) Constructs a sparse matrix representation of *values* and filters out certain

indices that are expected to contain more than *threshold* proportion of unlabeled reads.

2) Runs an EM algorithm that iteratively updates the filtered out data and stimation.

See https://doi.org/10.1093/bioinformatics/bty256.

Parameters

values
array of three columns encoding a sparse array in (row, column, value) format, zero-indexed, where row: number of conversions column: nucleotide content value: number of reads

p_e
Background mutation rate of unlabeled RNA

p_c
Initial p_c value

threshold
Filter threshold

max_iters
Maximum number of EM iterations

Returns
Estimated conversion rate

dynast.estimation.p_c.**estimate_p_c**(*df_aggregates: pandas.DataFrame*, *p_e: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]*, *p_c_path: str*, *group_by: Optional[List[str]] = None*, *threshold: int = 1000*, *n_threads: int = 8*, *nasc: bool = False*) → str

Estimate the average conversion rate in labeled RNA.

Parameters

df_aggregates
Pandas dataframe containing aggregate values

p_e
Background mutation rate of unlabeled RNA

p_c_path
Path to output CSV containing p_c estimates

group_by
Columns to group by

threshold
Read count threshold

n_threads
Number of threads

nasc
Flag to indicate whether to use NASC-seq pipeline variant of the EM algorithm

Returns
Path to output CSV containing p_c estimates

dynast.estimation.p_e

Module Contents

Functions

read_p_e(p_e_path: str, group_by: Optional[List[str]] = None) → Dict[Union[str, Tuple[str, Ellipsis]], float]	Read p_e CSV as a dictionary, with *group_by* columns as keys.
estimate_p_e_control(df_counts: pandas.DataFrame, p_e_path: str, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})) → str	Estimate background mutation rate of unlabeled RNA for a control sample
estimate_p_e(df_counts: pandas.DataFrame, p_e_path: str, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})}), group_by: Optional[List[str]] = None) → str	Estimate background mutation rate of unabeled RNA by calculating the
estimate_p_e_nasc(df_rates: pandas.DataFrame, p_e_path: str, group_by: Optional[List[str]] = None) → str	Estimate background mutation rate of unabeled RNA by calculating the

dynast.estimation.p_e.**read_p_e**(*p_e_path: str*, *group_by: Optional[List[str]] = None*) → Dict[Union[str, Tuple[str, Ellipsis]], float]

Read p_e CSV as a dictionary, with *group_by* columns as keys.

Parameters

p_e_path
Path to CSV containing p_e values

group_by
Columns to group by

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.p_e.**estimate_p_e_control**(*df_counts: pandas.DataFrame*, *p_e_path: str*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*) → str

Estimate background mutation rate of unlabeled RNA for a control sample by simply calculating the average mutation rate.

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

conversions
Conversion(s) in question

Returns
Path to output CSV containing p_e estimates

dynast.estimation.p_e.**estimate_p_e**(*df_counts: pandas.DataFrame*, *p_e_path: str*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*, *group_by: Optional[List[str]] = None*) → str

Estimate background mutation rate of unabeled RNA by calculating the average mutation rate of all three nucleotides other than *conversion[0]*.

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

conversions
Conversion(s) in question, defaults to *frozenset([('TC',)])*

group_by
Columns to group by, defaults to *None*

Returns
Path to output CSV containing p_e estimates

dynast.estimation.p_e.**estimate_p_e_nasc**(*df_rates: pandas.DataFrame*, *p_e_path: str*, *group_by: Optional[List[str]] = None*) → str

Estimate background mutation rate of unabeled RNA by calculating the average *CT* and *GA* mutation rates. This function imitates the procedure implemented in the NASC-seq pipeline (DOI: 10.1038/s41467-019-11028-9).

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

group_by
Columns to group by, defaults to *None*

Returns
Path to output CSV containing p_e estimates

dynast.estimation.pi

Module Contents

Functions

read_pi(pi_path: str, group_by: Optional[List[str]] = None) → Tuple[Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]]	Read pi CSV as a dictionary.
initializer(model: pystan.StanModel)	Multiprocessing initializer.
beta_mean(alpha: float, beta: float) → float	Calculate the mean of a beta distribution.
beta_mode(alpha: float, beta: float) → float	Calculate the mode of a beta distribution.
guess_beta_parameters(guess: float, strength: int = 5) → Tuple[float, float]	Given a *guess* of the mean of a beta distribution, calculate beta
fit_stan_mcmc(values: numpy.ndarray, p_e: float, p_c: float, guess: float = 0.5, model: pystan.StanModel = None, n_chains: int = 1, n_warmup: int = 1000, n_iters: int = 1000, n_threads: int = 1, seed: Optional[int] = None) → Tuple[float, float, float, float]	Run MCMC to estimate the fraction of labeled RNA.
estimate_pi(df_aggregates: pandas.DataFrame, p_e: float, p_c: float, pi_path: str, group_by: Optional[List[str]] = None, p_group_by: Optional[List[str]] = None, n_threads: int = 8, threshold: int = 16, seed: Optional[int] = None, nasc: bool = False, model: Optional[pystan.StanModel] = None) → str	Estimate the fraction of labeled RNA.

Attributes

_model

dynast.estimation.pi.**read_pi**(*pi_path: str*, *group_by: Optional[List[str]] = None*) → Tuple[Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]]

Read pi CSV as a dictionary.

Parameters

pi_path
path to CSV containing pi values

group_by
columns that were used to group estimation

Returns
Dictionary with barcodes and genes as keys

dynast.estimation.pi._model

dynast.estimation.pi.**initializer**(*model: pystan.StanModel*)

Multiprocessing initializer. https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor

This initializer performs a one-time expensive initialization for each process.

dynast.estimation.pi.**beta_mean**(*alpha: float*, *beta: float*) → float

Calculate the mean of a beta distribution. https://en.wikipedia.org/wiki/Beta_distribution

Parameters

alpha
First parameter of the beta distribution

beta
Second parameter of the beta distribution

Returns
Mean of the beta distribution

dynast.estimation.pi.**beta_mode**(*alpha: float*, *beta: float*) → float

Calculate the mode of a beta distribution. https://en.wikipedia.org/wiki/Beta_distribution

When the distribution is bimodal (*alpha*, *beta* < 1), this function returns *nan*.

Parameters

alpha
First parameter of the beta distribution

beta
Second parameter of the beta distribution

Returns
Mode of the beta distribution

dynast.estimation.pi.**guess_beta_parameters**(*guess: float*, *strength: int = 5*) → Tuple[float, float]

Given a *guess* of the mean of a beta distribution, calculate beta distribution parameters such that the distribution is skewed by some *strength* toward the *guess*.

Parameters

guess
Guess of the mean of the beta distribution

strength
Strength of the skew

Returns
Beta distribution parameters (alpha, beta)

dynast.estimation.pi.**fit_stan_mcmc**(*values: numpy.ndarray*, *p_e: float*, *p_c: float*, *guess: float = 0.5*, *model: pystan.StanModel = None*, *n_chains: int = 1*, *n_warmup: int = 1000*, *n_iters: int = 1000*, *n_threads: int = 1*, *seed: Optional[int] = None*) → Tuple[float, float, float, float]

Run MCMC to estimate the fraction of labeled RNA.

Parameters

values
Array of three columns encoding a sparse array in (row, column, value) format, zero-indexed, where row: number of conversions column: nucleotide content value: number of reads

p_e
Average mutation rate in unlabeled RNA

p_c
Average mutation rate in labeled RNA

guess
Guess for the fraction of labeled RNA

model
PyStan model to run MCMC with. If not provided, will try to use the *_model* global variable

n_chains
Number of MCMC chains

n_warmup
Number of warmup iterations

n_iters
Number of MCMC iterations, excluding any warmups

n_threads
Number of threads to use

seed
random seed used for MCMC

Returns
(guess, alpha, beta, pi)

dynast.estimation.pi.**estimate_pi**(*df_aggregates: pandas.DataFrame*, *p_e: float*, *p_c: float*, *pi_path: str*, *group_by: Optional[List[str]] = None*, *p_group_by: Optional[List[str]] = None*, *n_threads: int = 8*, *threshold: int = 16*, *seed: Optional[int] = None*, *nasc: bool = False*, *model: Optional[pystan.StanModel] = None*) → str

Estimate the fraction of labeled RNA.

Parameters

df_aggregates
Pandas dataframe containing aggregate values

p_e
Average mutation rate in unlabeled RNA

p_c
Average mutation rate in labeled RNA

pi_path
Path to write pi estimates

group_by
Columns that were used to group cells

p_group_by
Columns that p_e/p_c estimation was grouped by

n_threads
Number of threads

threshold
Any conversion-content pairs with fewer than this many reads will not be processed

seed
Random seed

nasc
Flag to change behavior to match NASC-seq pipeline. Specifically, the mode of the estimated Beta distribution is used as pi, defaults to *False*

model
PyStan model to run MCMC with. If not provided, will try to compile the module manually

Returns
Path to pi output

Package Contents

Functions

estimate_alpha(df_counts: pandas.DataFrame, pi_c: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], alpha_path: str, conversions: FrozenSet[str] = frozenset({'TC'}), group_by: Optional[List[str]] = None, pi_c_group_by: Optional[List[str]] = None) → str	Estimate the detection rate alpha.
read_alpha(alpha_path: str, group_by: Optional[List[str]] = None) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]	Read alpha CSV as a dictionary, with *group_by* columns as keys.
estimate_p_c(df_aggregates: pandas.DataFrame, p_e: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], p_c_path: str, group_by: Optional[List[str]] = None, threshold: int = 1000, n_threads: int = 8, nasc: bool = False) → str	Estimate the average conversion rate in labeled RNA.
read_p_c(p_c_path: str, group_by: Optional[List[str]] = None) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]	Read p_c CSV as a dictionary, with *group_by* columns as keys.
estimate_p_e(df_counts: pandas.DataFrame, p_e_path: str, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})}), group_by: Optional[List[str]] = None) → str	Estimate background mutation rate of unabeled RNA by calculating the
estimate_p_e_control(df_counts: pandas.DataFrame, p_e_path: str, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})) → str	Estimate background mutation rate of unlabeled RNA for a control sample
estimate_p_e_nasc(df_rates: pandas.DataFrame, p_e_path: str, group_by: Optional[List[str]] = None) → str	Estimate background mutation rate of unabeled RNA by calculating the
read_p_e(p_e_path: str, group_by: Optional[List[str]] = None) → Dict[Union[str, Tuple[str, Ellipsis]], float]	Read p_e CSV as a dictionary, with *group_by* columns as keys.
estimate_pi(df_aggregates: pandas.DataFrame, p_e: float, p_c: float, pi_path: str, group_by: Optional[List[str]] = None, p_group_by: Optional[List[str]] = None, n_threads: int = 8, threshold: int = 16, seed: Optional[int] = None, nasc: bool = False, model: Optional[pystan.StanModel] = None) → str	Estimate the fraction of labeled RNA.
read_pi(pi_path: str, group_by: Optional[List[str]] = None) → Tuple[Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]]	Read pi CSV as a dictionary.

dynast.estimation.**estimate_alpha**(*df_counts: pandas.DataFrame*, *pi_c: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]*, *alpha_path: str*, *conversions: FrozenSet[str] = frozenset({'TC'})*, *group_by: Optional[List[str]] = None*, *pi_c_group_by: Optional[List[str]] = None*) → str

Estimate the detection rate alpha.

Parameters

df_counts
Pandas dataframe containing conversion counts

pi_c
Labeled mutation rate

alpha_path
Path to output CSV containing alpha estimates

conversions
Conversions to consider

group_by
Columns to group by

pi_c_group_by
Columns that were used to group when calculating pi_c

Returns
Path to output CSV containing alpha estimates

dynast.estimation.**read_alpha**(*alpha_path: str*, *group_by: Optional[List[str]] = None*) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]

Read alpha CSV as a dictionary, with *group_by* columns as keys.

Parameters

alpha_path
Path to CSV containing alpha values

group_by
Columns to group by, defaults to *None*

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.**estimate_p_c**(*df_aggregates: pandas.DataFrame*, *p_e: Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]*, *p_c_path: str*, *group_by: Optional[List[str]] = None*, *threshold: int = 1000*, *n_threads: int = 8*, *nasc: bool = False*) → str

Estimate the average conversion rate in labeled RNA.

Parameters

df_aggregates
Pandas dataframe containing aggregate values

p_e
Background mutation rate of unlabeled RNA

p_c_path
Path to output CSV containing p_c estimates

group_by
Columns to group by

threshold
Read count threshold

n_threads
Number of threads

nasc
Flag to indicate whether to use NASC-seq pipeline variant of the EM algorithm

Returns
Path to output CSV containing p_c estimates

dynast.estimation.**read_p_c**(*p_c_path: str*, *group_by: Optional[List[str]] = None*) → Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]

Read p_c CSV as a dictionary, with *group_by* columns as keys.

Parameters

p_c_path
Path to CSV containing p_c values

group_by
Columns to group by, defaults to *None*

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.**estimate_p_e**(*df_counts: pandas.DataFrame*, *p_e_path: str*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*, *group_by: Optional[List[str]] = None*) → str

Estimate background mutation rate of unabeled RNA by calculating the average mutation rate of all three nucleotides other than *conversion[0]*.

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

conversions
Conversion(s) in question, defaults to *frozenset([('TC',)])*

group_by
Columns to group by, defaults to *None*

Returns
Path to output CSV containing p_e estimates

dynast.estimation.**estimate_p_e_control**(*df_counts: pandas.DataFrame*, *p_e_path: str*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*) → str

Estimate background mutation rate of unlabeled RNA for a control sample by simply calculating the average mutation rate.

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

conversions
Conversion(s) in question

Returns
Path to output CSV containing p_e estimates

dynast.estimation.**estimate_p_e_nasc**(*df_rates: pandas.DataFrame*, *p_e_path: str*, *group_by: Optional[List[str]] = None*) → str

Estimate background mutation rate of unabeled RNA by calculating the average *CT* and *GA* mutation rates. This function imitates the procedure implemented in the NASC-seq pipeline (DOI: 10.1038/s41467-019-11028-9).

Parameters

df_counts
Pandas dataframe containing number of each conversion and nucleotide content of each read

p_e_path
Path to output CSV containing p_e estimates

group_by
Columns to group by, defaults to *None*

Returns
Path to output CSV containing p_e estimates

dynast.estimation.**read_p_e**(*p_e_path: str*, *group_by: Optional[List[str]] = None*) → Dict[Union[str, Tuple[str, Ellipsis]], float]

Read p_e CSV as a dictionary, with *group_by* columns as keys.

Parameters

p_e_path
Path to CSV containing p_e values

group_by
Columns to group by

Returns
Dictionary with *group_by* columns as keys (tuple if multiple)

dynast.estimation.**estimate_pi**(*df_aggregates: pandas.DataFrame*, *p_e: float*, *p_c: float*, *pi_path: str*, *group_by: Optional[List[str]] = None*, *p_group_by: Optional[List[str]] = None*, *n_threads: int = 8*, *threshold: int = 16*, *seed: Optional[int] = None*, *nasc: bool = False*, *model: Optional[pystan.StanModel] = None*) → str

Estimate the fraction of labeled RNA.

Parameters

df_aggregates
Pandas dataframe containing aggregate values

p_e
Average mutation rate in unlabeled RNA

p_c
Average mutation rate in labeled RNA

pi_path
Path to write pi estimates

group_by
Columns that were used to group cells

p_group_by
Columns that p_e/p_c estimation was grouped by

n_threads
Number of threads

threshold
Any conversion-content pairs with fewer than this many reads will not be processed

seed
Random seed

nasc
Flag to change behavior to match NASC-seq pipeline. Specifically, the mode of the estimated Beta distribution is used as pi, defaults to *False*

model
PyStan model to run MCMC with. If not provided, will try to compile the module manually

Returns
Path to pi output

dynast.estimation.**read_pi**(*pi_path: str*, *group_by: Optional[List[str]] = None*) → Tuple[Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]], Union[float, Dict[str, float], Dict[Tuple[str, Ellipsis], float]]]

Read pi CSV as a dictionary.

Parameters

pi_path
path to CSV containing pi values

group_by
columns that were used to group estimation

Returns
Dictionary with barcodes and genes as keys

4.1.3 dynast.preprocessing

Submodules

dynast.preprocessing.aggregation

Module Contents

Functions

read_rates(rates_path: str) → pandas.DataFrame	Read mutation rates CSV as a pandas dataframe.
read_aggregates(aggregates_path: str) → pandas.DataFrame	Read aggregates CSV as a pandas dataframe.
merge_aggregates(*dfs: pandas.DataFrame) → pandas.DataFrame	Merge multiple aggregate dataframes into one.
calculate_mutation_rates(df_counts: pandas.DataFrame, rates_path: str, group_by: Optional[List[str]] = None) → str	Calculate mutation rate for each pair of bases.
aggregate_counts(df_counts: pandas.DataFrame, aggregates_path: str, conversions: FrozenSet[str] = frozenset({'TC'})) → str	Aggregate conversion counts for each pair of bases.

dynast.preprocessing.aggregation.**read_rates**(*rates_path: str*) → pandas.DataFrame

Read mutation rates CSV as a pandas dataframe.

Parameters

rates_path
Path to rates CSV

Returns
Rates dataframe

dynast.preprocessing.aggregation.**read_aggregates**(*aggregates_path: str*) → pandas.DataFrame

Read aggregates CSV as a pandas dataframe.

Parameters

aggregates_path
Path to aggregates CSV

Returns
Aggregates dataframe

dynast.preprocessing.aggregation.**merge_aggregates**(**dfs: pandas.DataFrame*) → pandas.DataFrame

Merge multiple aggregate dataframes into one.

Parameters

dfs
Dataframes to merge

Returns
Merged dataframe

dynast.preprocessing.aggregation.**calculate_mutation_rates**(*df_counts: pandas.DataFrame*, *rates_path: str*, *group_by: Optional[List[str]] = None*) → str

Calculate mutation rate for each pair of bases.

Parameters

df_counts
Counts dataframe, with complemented reverse strand bases

rates_path
Path to write rates CSV

group_by
Column(s) to group calculations by, defaults to *None*, which combines all rows

Returns
Path to rates CSV

dynast.preprocessing.aggregation.**aggregate_counts**(*df_counts: pandas.DataFrame*, *aggregates_path: str*, *conversions: FrozenSet[str] = frozenset({'TC'})*) → str

Aggregate conversion counts for each pair of bases.

Parameters

df_counts
Counts dataframe, with complemented reverse strand bases

aggregates_path
Path to write aggregate CSV

conversions
Conversion(s) in question

Returns
Path to aggregate CSV that was written

dynast.preprocessing.bam

Module Contents

Functions

read_alignments(alignments_path: str, *args, **kwargs) → pandas.DataFrame	Read alignments CSV as a pandas DataFrame.
read_conversions(conversions_path: str, *args, **kwargs) → pandas.DataFrame	Read conversions CSV as a pandas DataFrame.
select_alignments(df_alignments: pandas.DataFrame) → Set[Tuple[str, str]]	Select alignments among duplicates. This function performs preliminary
parse_read_contig(counter: multiprocessing.Value, lock: multiprocessing.Lock, bam_path: str, contig: str, gene_infos: Optional[dict] = None, transcript_infos: Optional[dict] = None, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, temp_dir: Optional[str] = None, update_every: int = 2000, nasc: bool = False, velocity: bool = True, strict_exon_overlap: bool = False) → Tuple[str, str, str]	Parse all reads mapped to a contig, outputing conversion
get_tags_from_bam(bam_path: str, n_reads: int = 100000, n_threads: int = 8) → Set[str]	Utility function to retrieve all read tags present in a BAM.
check_bam_tags_exist(bam_path: str, tags: List[str], n_reads: int = 100000, n_threads: int = 8) → Tuple[bool, List[str]]	Utility function to check if BAM tags exists in a BAM within the first
check_bam_is_paired(bam_path: str, n_reads: int = 100000, n_threads: int = 8) → bool	Utility function to check if BAM has paired reads.
check_bam_contains_secondary(bam_path: str, n_reads: int = 100000, n_threads: int = 8) → bool	Check whether BAM contains secondary alignments.
check_bam_contains_unmapped(bam_path: str) → bool	Check whether BAM contains unmapped reads.
check_bam_contains_duplicate(bam_path, n_reads=100000, n_threads=8) → bool	Check whether BAM contains duplicates.
sort_and_index_bam(bam_path: str, out_path: str, n_threads: int = 8, temp_dir: Optional[str] = None) → str	Sort and index BAM.
split_bam(bam_path: str, n: int, n_threads: int = 8, temp_dir: Optional[str] = None) → List[Tuple[str, int]]	Split BAM into n parts.
parse_all_reads(bam_path: str, conversions_path: str, alignments_path: str, index_path: str, gene_infos: dict, transcript_infos: dict, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, n_threads: int = 8, temp_dir: Optional[str] = None, nasc: bool = False, control: bool = False, velocity: bool = True, strict_exon_overlap: bool = False, return_splits: bool = False) → Union[Tuple[str, str, str], Tuple[str, str, str, List[Tuple[str, int]]]]	Parse all reads in a BAM and extract conversion, content and alignment

Attributes

CONVERSION_CSV_COLUMNS

ALIGNMENT_COLUMNS

dynast.preprocessing.bam.**CONVERSION_CSV_COLUMNS = ['read_id', 'index', 'contig', 'genome_i', 'conversion', 'quality']**

dynast.preprocessing.bam.**ALIGNMENT_COLUMNS = ['read_id', 'index', 'barcode', 'umi', 'GX', 'A', 'C', 'G', 'T', 'velocity', 'transcriptome', 'score']**

dynast.preprocessing.bam.**read_alignments**(*alignments_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read alignments CSV as a pandas DataFrame.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

alignments_path
path to alignments CSV

Returns
Conversions dataframe

dynast.preprocessing.bam.**read_conversions**(*conversions_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read conversions CSV as a pandas DataFrame.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

conversions_path
Path to conversions CSV

Returns
Conversions dataframe

dynast.preprocessing.bam.**select_alignments**(*df_alignments: pandas.DataFrame*) → Set[Tuple[str, str]]

Select alignments among duplicates. This function performs preliminary deduplication and returns a list of tuples (read_id, alignment index) to use for coverage calculation and SNP detection.

Parameters

df_alignments
Alignments dataframe

Returns
Set of (read_id, alignment index) that were selected

dynast.preprocessing.bam.**parse_read_contig**(*counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *bam_path: str*, *contig: str*, *gene_infos: Optional[dict] = None*, *transcript_infos: Optional[dict] = None*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *temp_dir: Optional[str] = None*, *update_every: int = 2000*, *nasc: bool = False*, *velocity: bool = True*, *strict_exon_overlap: bool = False*) → Tuple[str, str, str]

Parse all reads mapped to a contig, outputing conversion information as temporary CSVs. This function is designed to be called as a separate process.

Parameters

counter
Counter that keeps track of how many reads have been processed

lock
Semaphore for the *counter* so that multiple processes do not modify it at the same time

bam_path
Path to alignment BAM file

contig
Only reads that map to this contig will be processed

gene_infos
Dictionary containing gene information, as returned by *preprocessing.gtf.parse_gtf*, required if *velocity=True*

transcript_infos
Dictionary containing transcript information, as returned by *preprocessing.gtf.parse_gtf*, required if *velocity=True*

strand
Strandedness of the sequencing protocol, defaults to *forward*, may be one of the following: *forward*, *reverse*, *unstranded*

umi_tag
BAM tag that encodes UMI, if not provided, *NA* is output in the *umi* column

barcode_tag
BAM tag that encodes cell barcode, if not provided, *NA* is output in the *barcode* column

gene_tag
BAM tag that encodes gene assignment

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

temp_dir
Path to temporary directory

update_every
Update the counter every this many reads, defaults to *5000*

nasc
Flag to change behavior to match NASC-seq pipeline, defaults to *False*

velocity
Whether or not to assign a velocity type to each read

strict_exon_overlap
Whether to use a stricter algorithm to assin reads as spliced

Returns
(path to conversions, path to conversions index, path to alignments)

dynast.preprocessing.bam.**get_tags_from_bam**(*bam_path: str*, *n_reads: int = 100000*, *n_threads: int = 8*) → Set[str]

Utility function to retrieve all read tags present in a BAM.

Parameters

bam_path
Path to BAM

n_reads
Number of reads to consider

n_threads
Number of threads

Returns
Set of all tags found

dynast.preprocessing.bam.**check_bam_tags_exist**(*bam_path: str*, *tags: List[str]*, *n_reads: int = 100000*, *n_threads: int = 8*) → Tuple[bool, List[str]]

Utility function to check if BAM tags exists in a BAM within the first *n_reads* reads.

Parameters

bam_path
Path to BAM

tags
Tags to check for

n_reads
Number of reads to consider

n_threads
Number of threads

Returns
(whether all tags were found, list of not found tags)

dynast.preprocessing.bam.**check_bam_is_paired**(*bam_path: str*, *n_reads: int = 100000*, *n_threads: int = 8*) → bool

Utility function to check if BAM has paired reads.

bam_path: Path to BAM n_reads: Number of reads to consider n_threads: Number of threads

Returns
Whether paired reads were detected

dynast.preprocessing.bam.**check_bam_contains_secondary**(*bam_path: str*, *n_reads: int = 100000*, *n_threads: int = 8*) → bool

Check whether BAM contains secondary alignments.

bam_path: Path to BAM n_reads: Number of reads to consider n_threads: Number of threads

Returns
Whether secondary alignments were detected

dynast.preprocessing.bam.**check_bam_contains_unmapped**(*bam_path: str*) → bool

Check whether BAM contains unmapped reads.

bam_path: Path to BAM

Returns
Whether unmapped reads were detected

dynast.preprocessing.bam.**check_bam_contains_duplicate**(*bam_path*, *n_reads=100000*, *n_threads=8*) → bool

Check whether BAM contains duplicates.

bam_path: Path to BAM n_reads: Number of reads to consider n_threads: Number of threads

Returns
Whether duplicates were detected

dynast.preprocessing.bam.**sort_and_index_bam**(*bam_path: str*, *out_path: str*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*) → str

Sort and index BAM.

If the BAM is already sorted, the sorting step is skipped.

Parameters

bam_path
Path to alignment BAM file to sort

out_path
Path to output sorted BAM

n_threads
Number of threads

temp_dir
Path to temporary directory

Returns
Path to sorted and indexed BAM

dynast.preprocessing.bam.**split_bam**(*bam_path: str*, *n: int*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*) → List[Tuple[str, int]]

Split BAM into n parts.

Parameters

bam_path
Path to alignment BAM file

n
Number of splits

n_threads
Number of threads

temp_dir
Path to temporary directory

Returns
List of tuples containing (split BAM path, number of reads)

dynast.preprocessing.bam.**parse_all_reads**(*bam_path: str*, *conversions_path: str*, *alignments_path: str*, *index_path: str*, *gene_infos: dict*, *transcript_infos: dict*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *nasc: bool = False*, *control: bool = False*, *velocity: bool = True*, *strict_exon_overlap: bool = False*, *return_splits: bool = False*) → Union[Tuple[str, str, str], Tuple[str, str, str, List[Tuple[str, int]]]]

Parse all reads in a BAM and extract conversion, content and alignment information as CSVs.

bam_path: Path to alignment BAM file conversions_path: Path to output information about reads that have conversions alignments_path: Path to alignments information about reads index_path: Path to conversions index no_index_path: Path to no conversions index gene_infos: Dictionary containing gene information, as returned by

ngs.gtf.genes_and_transcripts_from_gtf

transcript_infos: Dictionary containing transcript information,
as returned by *ngs.gtf.genes_and_transcripts_from_gtf*

strand: Strandedness of the sequencing protocol, defaults to *forward*,
may be one of the following: *forward*, *reverse*, *unstranded*

umi_tag: BAM tag that encodes UMI, if not provided, *NA* is output in the
umi column

barcode_tag: BAM tag that encodes cell barcode, if not provided, *NA*
is output in the *barcode* column

gene_tag: BAM tag that encodes gene assignment, defaults to *GX* barcodes: List of barcodes to be considered. All barcodes are considered

if not provided

n_threads: Number of threads temp_dir: Path to temporary directory nasc: Flag to change behavior to match NASC-seq pipeline velocity: Whether or not to assign a velocity type to each read strict_exon_overlap: Whether to use a stricter algorithm to assign reads as spliced return_splits: return BAM splits for later reuse

Returns

(path to conversions, path to alignments, path to conversions index)
If *return_splits* is True, then there is an additional return value, which is a list of tuples containing split BAM paths and number of reads in each BAM.

dynast.preprocessing.consensus

Module Contents

Functions

call_consensus_from_reads(reads: List[pysam.AlignedSegment], header: pysam.AlignmentHeader, quality: int = 27, tags: Optional[Dict[str, Any]] = None) → pysam.AlignedSegment	Call a single consensus alignment given a list of aligned reads.
call_consensus_from_reads_process(reads, header, tags, strand=None, quality=27)	Helper function to call *call_consensus_from_reads()* from a subprocess.
consensus_worker(args_q, results_q, *args, **kwargs)	Multiprocessing worker.
call_consensus(bam_path: str, out_path: str, gene_infos: dict, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, quality: int = 27, add_RS_RI: bool = False, temp_dir: Optional[str] = None, n_threads: int = 8) → str	Call consensus sequences from BAM.

Attributes

BASES
BASE_IDX

dynast.preprocessing.consensus.**BASES = ['A', 'C', 'G', 'T']**

dynast.preprocessing.consensus.**BASE_IDX**

dynast.preprocessing.consensus.**call_consensus_from_reads**(*reads: List[pysam.AlignedSegment]*, *header: pysam.AlignmentHeader*, *quality: int = 27*, *tags: Optional[Dict[str, Any]] = None*) → pysam.AlignedSegment

Call a single consensus alignment given a list of aligned reads.

Reads must map to the same contig. Results are undefined otherwise. Additionally, consensus bases are called only for positions that match to the reference (i.e. no insertions allowed).

This function only sets the minimal amount of attributes such that the alignment is valid. These include: * read name – SHA256 hash of the provided read names * read sequence and qualities * reference name and ID * reference start * mapping quality (MAPQ) * cigarstring * MD tag * NM tag * Not unmapped, paired, duplicate, qc fail, secondary, nor supplementary

The caller is expected to further populate the alignment with additional tags, flags, and name.

Parameters

reads
List of reads to call a consensus sequence from

header
header to use when creating the new pysam alignment

quality
quality threshold

tags
additional tags to set

Returns
New pysam alignment of the consensus sequence

dynast.preprocessing.consensus.**call_consensus_from_reads_process**(*reads*, *header*, *tags*, *strand=None*, *quality=27*)

Helper function to call *call_consensus_from_reads()* from a subprocess.

dynast.preprocessing.consensus.**consensus_worker**(*args_q*, *results_q*, **args*, ***kwargs*)

Multiprocessing worker.

dynast.preprocessing.consensus.**call_consensus**(*bam_path: str*, *out_path: str*, *gene_infos: dict*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *quality: int = 27*, *add_RS_RI: bool = False*, *temp_dir: Optional[str] = None*, *n_threads: int = 8*) → str

Call consensus sequences from BAM.

Parameters

bam_path
Path to BAM

out_path
Output BAM path

gene_infos
Gene information, as parsed from the GTF

strand
Protocol strandedness

umi_tag
BAM tag containing the UMI

barcode_tag
BAM tag containing the barcode

gene_tag
BAM tag containing the assigned gene

barcodes
List of barcodes to consider

quality
Quality threshold

add_RS_RI
Add RS and RI BAM tags for debugging

temp_dir
Temporary directory

n_threads
Number of threads

Returns
Path to sorted and indexed consensus BAM

dynast.preprocessing.conversion

Module Contents

Functions

read_counts(counts_path: str, *args, **kwargs) → pandas.DataFrame	Read counts CSV as a pandas dataframe.
complement_counts(df_counts: pandas.DataFrame, gene_infos: dict) → pandas.DataFrame	Complement the counts in the counts dataframe according to gene strand.
subset_counts(df_counts: pandas.DataFrame, key: typing_extensions.Literal[total, transcriptome, spliced, unspliced]) → pandas.DataFrame	Subset the given counts DataFrame to only contain reads of the desired key.
drop_multimappers(df_counts: pandas.DataFrame, conversions: Optional[FrozenSet[str]] = None) → pandas.DataFrame	Drop multimappings that have the same read ID where
deduplicate_counts(df_counts: pandas.DataFrame, conversions: Optional[FrozenSet[str]] = None, use_conversions: bool = True) → pandas.DataFrame	Deduplicate counts based on barcode, UMI, and gene.
drop_multimappers_part(counter: multiprocessing.Value, lock: multiprocessing.Lock, split_path: str, out_path: str, conversions: Optional[FrozenSet[str]] = None) → str	Helper function to parallelize *drop_multimappers()*.
deduplicate_counts_part(counter: multiprocessing.Value, lock: multiprocessing.Lock, split_path: str, out_path: str, conversions: Optional[FrozenSet[str]], use_conversions: bool = True)	Helper function to parallelize `deduplicate_multimappers()`.
split_counts_by_velocity(df_counts: pandas.DataFrame) → Dict[str, pandas.DataFrame]	Split the given counts dataframe by the *velocity* column.
count_no_conversions(alignments_path: str, counter: multiprocessing.Value, lock: multiprocessing.Lock, index: List[Tuple[int, int, int]], barcodes: Optional[List[str]] = None, temp_dir: Optional[str] = None, update_every: int = 10000) → str	Count reads that have no conversion.
count_conversions_part(conversions_path: str, alignments_path: str, counter: multiprocessing.Value, lock: multiprocessing.Lock, index: List[Tuple[int, int, int]], barcodes: Optional[List[str]] = None, snps: Optional[Dict[str, Dict[str, Set[int]]]] = None, quality: int = 27, temp_dir: Optional[str] = None, update_every: int = 10000) → str	Count the number of conversions of each read per barcode and gene, along with
count_conversions(conversions_path: str, alignments_path: str, index_path: str, counts_path: str, gene_infos: dict, barcodes: Optional[List[str]] = None, snps: Optional[Dict[str, Dict[str, Set[int]]]] = None, quality: int = 27, conversions: Optional[FrozenSet[str]] = None, dedup_use_conversions: bool = True, n_threads: int = 8, temp_dir: Optional[str] = None) → str	Count the number of conversions of each read per barcode and gene, along with

Attributes

CONVERSIONS_PARSER
ALIGNMENTS_PARSER
CONVERSION_IDX
BASE_IDX
CONVERSION_COMPLEMENT
CONVERSION_COLUMNS
BASE_COLUMNS
COLUMNS
CSV_COLUMNS

dynast.preprocessing.conversion.**CONVERSIONS_PARSER**

dynast.preprocessing.conversion.**ALIGNMENTS_PARSER**

dynast.preprocessing.conversion.**CONVERSION_IDX**

dynast.preprocessing.conversion.**BASE_IDX**

dynast.preprocessing.conversion.**CONVERSION_COMPLEMENT**

dynast.preprocessing.conversion.**CONVERSION_COLUMNS**

dynast.preprocessing.conversion.**BASE_COLUMNS**

dynast.preprocessing.conversion.**COLUMNS**

dynast.preprocessing.conversion.**CSV_COLUMNS**

dynast.preprocessing.conversion.**read_counts**(*counts_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read counts CSV as a pandas dataframe.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

counts_path
Path to CSV

Returns
Counts dataframe

dynast.preprocessing.conversion.**complement_counts**(*df_counts: pandas.DataFrame*, *gene_infos: dict*) → pandas.DataFrame

Complement the counts in the counts dataframe according to gene strand.

Parameters

df_counts
Counts dataframe

gene_infos
Dictionary containing gene information, as returned by *preprocessing.gtf.parse_gtf*

Returns
counts dataframe with counts complemented for reads mapping to genes on the reverse strand

dynast.preprocessing.conversion.**subset_counts**(*df_counts: pandas.DataFrame*, *key: typing_extensions.Literal[total, transcriptome, spliced, unspliced]*) → pandas.DataFrame

Subset the given counts DataFrame to only contain reads of the desired key.

Parameters

df_count
Counts dataframe

key
Read types to subset

Returns:s
Subset dataframe

dynast.preprocessing.conversion.**drop_multimappers**(*df_counts: pandas.DataFrame*, *conversions: Optional[FrozenSet[str]] = None*) → pandas.DataFrame

Drop multimappings that have the same read ID where * some map to the transcriptome while some do not – drop non-transcriptome alignments * none map to the transcriptome AND aligned to multiple genes – drop all * none map to the transcriptome AND assigned multiple velocity types – set to ambiguous

TODO: This function can probably be removed because BAM parsing only considers primary alignments now.

Parameters

df_counts
Counts dataframe

conversions
Conversions to prioritize

Returns
Counts dataframe with multimappers appropriately filtered

dynast.preprocessing.conversion.**deduplicate_counts**(*df_counts: pandas.DataFrame*, *conversions: Optional[FrozenSet[str]] = None*, *use_conversions: bool = True*) → pandas.DataFrame

Deduplicate counts based on barcode, UMI, and gene.

The order of priority is the following. 1. If *use_conversions=True*, reads that have at least one such conversion 2. Reads that align to the transcriptome (exon only) 3. Reads that have highest alignment score 4. If *conversions* is provided, reads that have a larger sum of such conversions

If *conversions* is not provided, reads that have larger sum of all conversions

Parameters

df_counts
Counts dataframe

conversions
Conversions to prioritize, defaults to *None*

use_conversions
Prioritize reads that have conversions first

Returns
Deduplicated counts dataframe

dynast.preprocessing.conversion.**drop_multimappers_part**(*counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *split_path: str*, *out_path: str*, *conversions: Optional[FrozenSet[str]] = None*) → str

Helper function to parallelize `drop_multimappers()`.

dynast.preprocessing.conversion.**deduplicate_counts_part**(*counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *split_path: str*, *out_path: str*, *conversions: Optional[FrozenSet[str]]*, *use_conversions: bool = True*)

Helper function to parallelize `deduplicate_multimappers()`.

dynast.preprocessing.conversion.**split_counts_by_velocity**(*df_counts: pandas.DataFrame*) → Dict[str, pandas.DataFrame]

Split the given counts dataframe by the *velocity* column.

Parameters

df_counts
Counts dataframe

Returns
Dictionary containing *velocity* column values as keys and the subset dataframe as values

dynast.preprocessing.conversion.**count_no_conversions**(*alignments_path: str*, *counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *index: List[Tuple[int, int, int]]*, *barcodes: Optional[List[str]] = None*, *temp_dir: Optional[str] = None*, *update_every: int = 10000*) → str

Count reads that have no conversion.

Parameters

alignments_path
Alignments CSV path

counter
Counter that keeps track of how many reads have been processed

lock
Semaphore for the *counter* so that multiple processes do not modify it at the same time

index
Index for conversions CSV

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

temp_dir
Path to temporary directory

update_every
Update the counter every this many reads

Returns
Path to temporary counts CSV

dynast.preprocessing.conversion.**count_conversions_part**(*conversions_path: str*, *alignments_path: str*, *counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *index: List[Tuple[int, int, int]]*, *barcodes: Optional[List[str]] = None*, *snps: Optional[Dict[str, Dict[str, Set[int]]]] = None*, *quality: int = 27*, *temp_dir: Optional[str] = None*, *update_every: int = 10000*) → str

Count the number of conversions of each read per barcode and gene, along with the total nucleotide content of the region each read mapped to, also per barcode and gene. This function is used exclusively for multiprocessing.

Parameters

conversions_path
Path to conversions CSV

alignments_path
Path to alignments information about reads

counter
Counter that keeps track of how many reads have been processed

lock
Semaphore for the *counter* so that multiple processes do not modify it at the same time

index
Index for conversions CSV

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

snps
Dictionary of contig as keys and list of genomic positions as values that indicate SNP locations

quality
Only count conversions with PHRED quality greater than this value

temp_dir
Path to temporary directory, defaults to *None*

update_every
Update the counter every this many reads

Returns
Path to temporary counts CSV

dynast.preprocessing.conversion.**count_conversions**(*conversions_path: str*, *alignments_path: str*, *index_path: str*, *counts_path: str*, *gene_infos: dict*, *barcodes: Optional[List[str]] = None*, *snps: Optional[Dict[str, Dict[str, Set[int]]]] = None*, *quality: int = 27*, *conversions: Optional[FrozenSet[str]] = None*, *dedup_use_conversions: bool = True*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*) → str

Count the number of conversions of each read per barcode and gene, along with the total nucleotide content of the region each read mapped to, also per barcode. When a duplicate UMI for a barcode is observed, the read with the greatest number of conversions is selected.

Parameters

conversions_path
Path to conversions CSV

alignments_path
Path to alignments information about reads

index_path
Path to conversions index

counts_path
Path to write counts CSV

gene_infos
Dictionary containing gene information, as returned by *ngs.gtf.genes_and_transcripts_from_gtf*

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

snps
Dictionary of contig as keys and list of genomic positions as values that indicate SNP locations

conversions
Conversions to prioritize when deduplicating only applicable for UMI technologies

dedup_use_conversions
Prioritize reads that have at least one conversion when deduplicating

quality
Only count conversions with PHRED quality greater than this value

n_threads
Number of threads

temp_dir
Path to temporary directory

Returns
Path to counts CSV

dynast.preprocessing.coverage

Module Contents

Functions

read_coverage(coverage_path: str) → Dict[str, Dict[int, int]]	Read coverage CSV as a dictionary.
calculate_coverage_contig(counter: multiprocessing.Value, lock: multiprocessing.Lock, bam_path: str, contig: str, indices: List[Tuple[int, int, int]], alignments: Set[Tuple[str, int]] = None, umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, temp_dir: Optional[str] = None, update_every: int = 50000, velocity: bool = True) → str	Calculate converage for a specific contig. This function is designed to
calculate_coverage(bam_path: str, conversions: Dict[str, Set[int]], coverage_path: str, alignments: Optional[List[Tuple[str, int]]] = None, umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, temp_dir: Optional[str] = None, velocity: bool = True) → str	Calculate coverage of each genomic position per barcode.

Attributes

COVERAGE_PARSER

dynast.preprocessing.coverage.**COVERAGE_PARSER**

dynast.preprocessing.coverage.**read_coverage**(*coverage_path: str*) → Dict[str, Dict[int, int]]

Read coverage CSV as a dictionary.

Parameters

coverage_path
Path to coverage CSV

Returns
Coverage as a nested dictionary

dynast.preprocessing.coverage.**calculate_coverage_contig**(*counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *bam_path: str*, *contig: str*, *indices: List[Tuple[int, int, int]]*, *alignments: Set[Tuple[str, int]] = None*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *temp_dir: Optional[str] = None*, *update_every: int = 50000*, *velocity: bool = True*) → str

Calculate converage for a specific contig. This function is designed to be called as a separate process.

Parameters

`counter`
Counter that keeps track of how many reads have been processed

`lock`
Semaphore for the *counter* so that multiple processes do not modify it at the same time

`bam_path`
Path to alignment BAM file

`contig`
Only reads that map to this contig will be processed

`indices`
Genomic positions to consider

`alignments`
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

`umi_tag`
BAM tag that encodes UMI, if not provided, *NA* is output in the *umi* column

`barcode_tag`
BAM tag that encodes cell barcode, if not provided, *NA* is output in the *barcode* column

`gene_tag`
BAM tag that encodes gene assignment, defaults to *GX*

`barcodes`
List of barcodes to be considered. All barcodes are considered if not provided

`temp_dir`
Path to temporary directory

`update_every`
Update the counter every this many reads

`velocity`
Whether or not velocities were assigned

Returns
Path to coverage CSV

`dynast.preprocessing.coverage.calculate_coverage`(*bam_path: str*, *conversions: Dict[str, Set[int]]*, *coverage_path: str*, *alignments: Optional[List[Tuple[str, int]]] = None*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *temp_dir: Optional[str] = None*, *velocity: bool = True*) → str

Calculate coverage of each genomic position per barcode.

Parameters

`bam_path`
Path to alignment BAM file

conversions
Dictionary of contigs as keys and sets of genomic positions as values that indicates positions where conversions were observed

coverage_path
Path to write coverage CSV

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

umi_tag
BAM tag that encodes UMI, if not provided, *NA* is output in the *umi* column

barcode_tag
BAM tag that encodes cell barcode, if not provided, *NA* is output in the *barcode* column

gene_tag
BAM tag that encodes gene assignment

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

temp_dir
Path to temporary directory

velocity
Whether or not velocities were assigned

Returns
Path to coverage CSV

dynast.preprocessing.snp

Module Contents

Functions

read_snps(snps_path: str) → Dict[str, Dict[str, Set[int]]]	Read SNPs CSV as a dictionary
read_snp_csv(snp_csv: str) → Dict[str, Dict[str, Set[int]]]	Read a user-provided SNPs CSV
extract_conversions_part(conversions_path: str, counter: multiprocessing.Value, lock: multiprocessing.Lock, index: List[Tuple[int, int, int]], alignments: Optional[List[Tuple[str, int]]] = None, conversions: Optional[FrozenSet[str]] = None, quality: int = 27, update_every: int = 5000) → Dict[str, Dict[str, Dict[int, int]]]	Extract number of conversions for every genomic position.
extract_conversions(conversions_path: str, index_path: str, alignments: Optional[List[Tuple[str, int]]] = None, conversions: Optional[FrozenSet[str]] = None, quality: int = 27, n_threads: int = 8) → Dict[str, Dict[str, Dict[int, int]]]	Wrapper around *extract_conversions_part* that works in parallel
detect_snps(conversions_path: str, index_path: str, coverage: Dict[str, Dict[int, int]], snps_path: str, alignments: Optional[List[Tuple[str, int]]] = None, conversions: Optional[FrozenSet[str]] = None, quality: int = 27, threshold: float = 0.5, min_coverage: int = 1, n_threads: int = 8) → str	Detect SNPs.

Attributes

SNP_COLUMNS

dynast.preprocessing.snp.**SNP_COLUMNS = ['contig', 'genome_i', 'conversion']**

dynast.preprocessing.snp.**read_snps**(*snps_path: str*) → Dict[str, Dict[str, Set[int]]]

Read SNPs CSV as a dictionary

Parameters

snps_path
Path to SNPs CSV

Returns
Dictionary of contigs as keys and sets of genomic positions with SNPs as values

dynast.preprocessing.snp.**read_snp_csv**(*snp_csv: str*) → Dict[str, Dict[str, Set[int]]]

Read a user-provided SNPs CSV

Parameters

snp_csv
Path to SNPs CSV

Returns
Dictionary of contigs as keys and sets of genomic positions with SNPs as values

dynast.preprocessing.snp.**extract_conversions_part**(*conversions_path: str*, *counter: multiprocessing.Value*, *lock: multiprocessing.Lock*, *index: List[Tuple[int, int, int]]*, *alignments: Optional[List[Tuple[str, int]]] = None*, *conversions: Optional[FrozenSet[str]] = None*, *quality: int = 27*, *update_every: int = 5000*) → Dict[str, Dict[str, Dict[int, int]]]

Extract number of conversions for every genomic position.

Parameters

conversions_path
Path to conversions CSV

counter
Counter that keeps track of how many reads have been processed

lock
Semaphore for the *counter* so that multiple processes do not modify it at the same time

index
Conversions index

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

conversions
Set of conversions to consider

quality
Only count conversions with PHRED quality greater than this value

update_every
Update the counter every this many reads

Returns
Nested dictionary that contains number of conversions for each contig and position

dynast.preprocessing.snp.**extract_conversions**(*conversions_path: str*, *index_path: str*, *alignments: Optional[List[Tuple[str, int]]] = None*, *conversions: Optional[FrozenSet[str]] = None*, *quality: int = 27*, *n_threads: int = 8*) → Dict[str, Dict[str, Dict[int, int]]]

Wrapper around *extract_conversions_part* that works in parallel

Parameters

conversions_path
Path to conversions CSV

index_path
Path to conversions index

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

conversions
Set of conversions to consider

quality
Only count conversions with PHRED quality greater than this value

n_threads
Number of threads

Returns
Nested dictionary that contains number of conversions for each contig and position

dynast.preprocessing.snp.**detect_snps**(*conversions_path: str*, *index_path: str*, *coverage: Dict[str, Dict[int, int]]*, *snps_path: str*, *alignments: Optional[List[Tuple[str, int]]] = None*, *conversions: Optional[FrozenSet[str]] = None*, *quality: int = 27*, *threshold: float = 0.5*, *min_coverage: int = 1*, *n_threads: int = 8*) → str

Detect SNPs.

Parameters

conversions_path
Path to conversions CSV

index_path
Path to conversions index

coverage
Dictionary containing genomic coverage

snps_path
Path to output SNPs

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

conversions
Set of conversions to consider

quality
Only count conversions with PHRED quality greater than this value

threshold
Positions with conversions / coverage > threshold will be considered as SNPs

min_coverage
Only positions with at least this many mapping read_snps are considered

n_threads
Number of threads

Returns
Path to SNPs CSV

Package Contents

Functions

aggregate_counts(df_counts: pandas.DataFrame, aggregates_path: str, conversions: FrozenSet[str] = frozenset({'TC'})) → str	Aggregate conversion counts for each pair of bases.
calculate_mutation_rates(df_counts: pandas.DataFrame, rates_path: str, group_by: Optional[List[str]] = None) → str	Calculate mutation rate for each pair of bases.
merge_aggregates(*dfs: pandas.DataFrame) → pandas.DataFrame	Merge multiple aggregate dataframes into one.
read_aggregates(aggregates_path: str) → pandas.DataFrame	Read aggregates CSV as a pandas dataframe.
read_rates(rates_path: str) → pandas.DataFrame	Read mutation rates CSV as a pandas dataframe.
check_bam_contains_duplicate(bam_path, n_reads=100000, n_threads=8) → bool	Check whether BAM contains duplicates.
check_bam_contains_secondary(bam_path: str, n_reads: int = 100000, n_threads: int = 8) → bool	Check whether BAM contains secondary alignments.
check_bam_contains_unmapped(bam_path: str) → bool	Check whether BAM contains unmapped reads.
get_tags_from_bam(bam_path: str, n_reads: int = 100000, n_threads: int = 8) → Set[str]	Utility function to retrieve all read tags present in a BAM.
parse_all_reads(bam_path: str, conversions_path: str, alignments_path: str, index_path: str, gene_infos: dict, transcript_infos: dict, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, n_threads: int = 8, temp_dir: Optional[str] = None, nasc: bool = False, control: bool = False, velocity: bool = True, strict_exon_overlap: bool = False, return_splits: bool = False) → Union[Tuple[str, str, str], Tuple[str, str, str, List[Tuple[str, int]]]]	Parse all reads in a BAM and extract conversion, content and alignment
read_alignments(alignments_path: str, *args, **kwargs) → pandas.DataFrame	Read alignments CSV as a pandas DataFrame.
read_conversions(conversions_path: str, *args, **kwargs) → pandas.DataFrame	Read conversions CSV as a pandas DataFrame.
select_alignments(df_alignments: pandas.DataFrame) → Set[Tuple[str, str]]	Select alignments among duplicates. This function performs preliminary
sort_and_index_bam(bam_path: str, out_path: str, n_threads: int = 8, temp_dir: Optional[str] = None) → str	Sort and index BAM.
call_consensus(bam_path: str, out_path: str, gene_infos: dict, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, quality: int = 27, add_RS_RI: bool = False, temp_dir: Optional[str] = None, n_threads: int = 8) → str	Call consensus sequences from BAM.
complement_counts(df_counts: pandas.DataFrame, gene_infos: dict) → pandas.DataFrame	Complement the counts in the counts dataframe according to gene strand.
count_conversions(conversions_path: str, alignments_path: str, index_path: str, counts_path: str, gene_infos: dict, barcodes: Optional[List[str]] = None, snps: Optional[Dict[str, Dict[str, Set[int]]]] = None, quality: int = 27, conversions: Optional[FrozenSet[str]] = None, dedup_use_conversions: bool = True, n_threads: int = 8, temp_dir: Optional[str] = None) → str	Count the number of conversions of each read per barcode and gene, along with

Attributes

CONVERSION_COMPLEMENT

dynast.preprocessing.**aggregate_counts**(*df_counts: pandas.DataFrame*, *aggregates_path: str*, *conversions: FrozenSet[str] = frozenset({'TC'})*) → str

Aggregate conversion counts for each pair of bases.

Parameters

df_counts
Counts dataframe, with complemented reverse strand bases

aggregates_path
Path to write aggregate CSV

conversions
Conversion(s) in question

Returns
Path to aggregate CSV that was written

dynast.preprocessing.**calculate_mutation_rates**(*df_counts: pandas.DataFrame*, *rates_path: str*, *group_by: Optional[List[str]] = None*) → str

Calculate mutation rate for each pair of bases.

Parameters

df_counts
Counts dataframe, with complemented reverse strand bases

rates_path
Path to write rates CSV

group_by
Column(s) to group calculations by, defaults to *None*, which combines all rows

Returns
Path to rates CSV

dynast.preprocessing.**merge_aggregates**(**dfs: pandas.DataFrame*) → pandas.DataFrame

Merge multiple aggregate dataframes into one.

Parameters

dfs
Dataframes to merge

Returns
Merged dataframe

dynast.preprocessing.**read_aggregates**(*aggregates_path: str*) → pandas.DataFrame

Read aggregates CSV as a pandas dataframe.

Parameters

aggregates_path
Path to aggregates CSV

Returns
Aggregates dataframe

dynast.preprocessing.**read_rates**(*rates_path: str*) → pandas.DataFrame

Read mutation rates CSV as a pandas dataframe.

Parameters

rates_path
Path to rates CSV

Returns
Rates dataframe

dynast.preprocessing.**check_bam_contains_duplicate**(*bam_path*, *n_reads=100000*, *n_threads=8*) → bool

Check whether BAM contains duplicates.

bam_path: Path to BAM n_reads: Number of reads to consider n_threads: Number of threads

Returns
Whether duplicates were detected

dynast.preprocessing.**check_bam_contains_secondary**(*bam_path: str*, *n_reads: int = 100000*, *n_threads: int = 8*) → bool

Check whether BAM contains secondary alignments.

bam_path: Path to BAM n_reads: Number of reads to consider n_threads: Number of threads

Returns
Whether secondary alignments were detected

dynast.preprocessing.**check_bam_contains_unmapped**(*bam_path: str*) → bool

Check whether BAM contains unmapped reads.

bam_path: Path to BAM

Returns
Whether unmapped reads were detected

dynast.preprocessing.**get_tags_from_bam**(*bam_path: str*, *n_reads: int = 100000*, *n_threads: int = 8*) → Set[str]

Utility function to retrieve all read tags present in a BAM.

Parameters

bam_path
Path to BAM

n_reads
Number of reads to consider

n_threads
Number of threads

Returns
Set of all tags found

dynast.preprocessing.**parse_all_reads**(*bam_path: str*, *conversions_path: str*, *alignments_path: str*, *index_path: str*, *gene_infos: dict*, *transcript_infos: dict*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *nasc: bool = False*, *control: bool = False*, *velocity: bool = True*, *strict_exon_overlap: bool = False*, *return_splits: bool = False*) → Union[Tuple[str, str, str], Tuple[str, str, str, List[Tuple[str, int]]]]

Parse all reads in a BAM and extract conversion, content and alignment information as CSVs.

bam_path: Path to alignment BAM file conversions_path: Path to output information about reads that have conversions alignments_path: Path to alignments information about reads index_path: Path to conversions index no_index_path: Path to no conversions index gene_infos: Dictionary containing gene information, as returned by

ngs.gtf.genes_and_transcripts_from_gtf

transcript_infos: Dictionary containing transcript information,
as returned by *ngs.gtf.genes_and_transcripts_from_gtf*

strand: Strandedness of the sequencing protocol, defaults to *forward*,
may be one of the following: *forward*, *reverse*, *unstranded*

umi_tag: BAM tag that encodes UMI, if not provided, *NA* is output in the
umi column

barcode_tag: BAM tag that encodes cell barcode, if not provided, *NA*
is output in the *barcode* column

gene_tag: BAM tag that encodes gene assignment, defaults to *GX* barcodes: List of barcodes to be considered. All barcodes are considered

if not provided

n_threads: Number of threads temp_dir: Path to temporary directory nasc: Flag to change behavior to match NASC-seq pipeline velocity: Whether or not to assign a velocity type to each read strict_exon_overlap: Whether to use a stricter algorithm to assign reads as spliced return_splits: return BAM splits for later reuse

Returns

(path to conversions, path to alignments, path to conversions index)
If *return_splits* is True, then there is an additional return value, which is a list of tuples containing split BAM paths and number of reads in each BAM.

dynast.preprocessing.**read_alignments**(*alignments_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read alignments CSV as a pandas DataFrame.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

alignments_path
path to alignments CSV

Returns
Conversions dataframe

dynast.preprocessing.**read_conversions**(*conversions_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read conversions CSV as a pandas DataFrame.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

conversions_path
Path to conversions CSV

Returns
Conversions dataframe

dynast.preprocessing.**select_alignments**(*df_alignments: pandas.DataFrame*) → Set[Tuple[str, str]]

Select alignments among duplicates. This function performs preliminary deduplication and returns a list of tuples (read_id, alignment index) to use for coverage calculation and SNP detection.

Parameters

df_alignments
Alignments dataframe

Returns
Set of (read_id, alignment index) that were selected

dynast.preprocessing.**sort_and_index_bam**(*bam_path: str*, *out_path: str*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*) → str

Sort and index BAM.

If the BAM is already sorted, the sorting step is skipped.

Parameters

bam_path
Path to alignment BAM file to sort

out_path
Path to output sorted BAM

n_threads
Number of threads

temp_dir
Path to temporary directory

Returns
Path to sorted and indexed BAM

dynast.preprocessing.**call_consensus**(*bam_path: str*, *out_path: str*, *gene_infos: dict*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *quality: int = 27*, *add_RS_RI: bool = False*, *temp_dir: Optional[str] = None*, *n_threads: int = 8*) → str

Call consensus sequences from BAM.

Parameters

bam_path
Path to BAM

out_path
Output BAM path

gene_infos
Gene information, as parsed from the GTF

strand
Protocol strandedness

umi_tag
BAM tag containing the UMI

barcode_tag
BAM tag containing the barcode

gene_tag
BAM tag containing the assigned gene

barcodes
List of barcodes to consider

quality
Quality threshold

add_RS_RI
Add RS and RI BAM tags for debugging

temp_dir
Temporary directory

n_threads
Number of threads

Returns
Path to sorted and indexed consensus BAM

dynast.preprocessing.**CONVERSION_COMPLEMENT**

dynast.preprocessing.**complement_counts**(*df_counts: pandas.DataFrame*, *gene_infos: dict*) → pandas.DataFrame

Complement the counts in the counts dataframe according to gene strand.

Parameters

df_counts
Counts dataframe

gene_infos
Dictionary containing gene information, as returned by *preprocessing.gtf.parse_gtf*

Returns
counts dataframe with counts complemented for reads mapping to genes on the reverse strand

dynast.preprocessing.**count_conversions**(*conversions_path: str*, *alignments_path: str*, *index_path: str*, *counts_path: str*, *gene_infos: dict*, *barcodes: Optional[List[str]] = None*, *snps: Optional[Dict[str, Dict[str, Set[int]]]] = None*, *quality: int = 27*, *conversions: Optional[FrozenSet[str]] = None*, *dedup_use_conversions: bool = True*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*) → str

Count the number of conversions of each read per barcode and gene, along with the total nucleotide content of the region each read mapped to, also per barcode. When a duplicate UMI for a barcode is observed, the read with the greatest number of conversions is selected.

Parameters

conversions_path
Path to conversions CSV

alignments_path
Path to alignments information about reads

index_path
Path to conversions index

counts_path
Path to write counts CSV

gene_infos
Dictionary containing gene information, as returned by *ngs.gtf.genes_and_transcripts_from_gtf*

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

snps
Dictionary of contig as keys and list of genomic positions as values that indicate SNP locations

conversions
Conversions to prioritize when deduplicating only applicable for UMI technologies

dedup_use_conversions
Prioritize reads that have at least one conversion when deduplicating

quality
Only count conversions with PHRED quality greater than this value

n_threads
Number of threads

temp_dir
Path to temporary directory

Returns
Path to counts CSV

dynast.preprocessing.**deduplicate_counts**(*df_counts: pandas.DataFrame*, *conversions: Optional[FrozenSet[str]] = None*, *use_conversions: bool = True*) → pandas.DataFrame

Deduplicate counts based on barcode, UMI, and gene.

The order of priority is the following. 1. If *use_conversions=True*, reads that have at least one such conversion 2. Reads that align to the transcriptome (exon only) 3. Reads that have highest alignment score 4. If *conversions* is provided, reads that have a larger sum of such conversions

If *conversions* is not provided, reads that have larger sum of all conversions

Parameters

df_counts
Counts dataframe

conversions
Conversions to prioritize, defaults to *None*

use_conversions
Prioritize reads that have conversions first

Returns
Deduplicated counts dataframe

dynast.preprocessing.**read_counts**(*counts_path: str*, **args*, ***kwargs*) → pandas.DataFrame

Read counts CSV as a pandas dataframe.

Any additional arguments and keyword arguments are passed to *pandas.read_csv*.

Parameters

counts_path
Path to CSV

Returns
Counts dataframe

dynast.preprocessing.**split_counts_by_velocity**(*df_counts: pandas.DataFrame*) → Dict[str, pandas.DataFrame]

Split the given counts dataframe by the *velocity* column.

Parameters

df_counts
Counts dataframe

Returns
Dictionary containing *velocity* column values as keys and the subset dataframe as values

dynast.preprocessing.**subset_counts**(*df_counts: pandas.DataFrame*, *key: typing_extensions.Literal[total, transcriptome, spliced, unspliced]*) → pandas.DataFrame

Subset the given counts DataFrame to only contain reads of the desired key.

Parameters

df_count
Counts dataframe

key
Read types to subset

Returns:s
Subset dataframe

dynast.preprocessing.**calculate_coverage**(*bam_path: str*, *conversions: Dict[str, Set[int]]*, *coverage_path: str*, *alignments: Optional[List[Tuple[str, int]]] = None*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *temp_dir: Optional[str] = None*, *velocity: bool = True*) → str

Calculate coverage of each genomic position per barcode.

Parameters

bam_path
Path to alignment BAM file

conversions
Dictionary of contigs as keys and sets of genomic positions as values that indicates positions where conversions were observed

coverage_path
Path to write coverage CSV

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

umi_tag
BAM tag that encodes UMI, if not provided, *NA* is output in the *umi* column

barcode_tag
BAM tag that encodes cell barcode, if not provided, *NA* is output in the *barcode* column

gene_tag
BAM tag that encodes gene assignment

barcodes
List of barcodes to be considered. All barcodes are considered if not provided

temp_dir
Path to temporary directory

velocity
Whether or not velocities were assigned

Returns
Path to coverage CSV

dynast.preprocessing.**read_coverage**(*coverage_path: str*) → Dict[str, Dict[int, int]]

Read coverage CSV as a dictionary.

Parameters

coverage_path
Path to coverage CSV

Returns
Coverage as a nested dictionary

dynast.preprocessing.**detect_snps**(*conversions_path: str*, *index_path: str*, *coverage: Dict[str, Dict[int, int]]*, *snps_path: str*, *alignments: Optional[List[Tuple[str, int]]] = None*, *conversions: Optional[FrozenSet[str]] = None*, *quality: int = 27*, *threshold: float = 0.5*, *min_coverage: int = 1*, *n_threads: int = 8*) → str

Detect SNPs.

Parameters

conversions_path
Path to conversions CSV

index_path
Path to conversions index

coverage
Dictionary containing genomic coverage

snps_path
Path to output SNPs

alignments
Set of (read_id, alignment_index) tuples to process. All alignments are processed if this option is not provided.

conversions
Set of conversions to consider

quality
Only count conversions with PHRED quality greater than this value

threshold
Positions with conversions / coverage > threshold will be considered as SNPs

min_coverage
Only positions with at least this many mapping read_snps are considered

n_threads
Number of threads

Returns
Path to SNPs CSV

dynast.preprocessing.**read_snp_csv**(*snp_csv: str*) → Dict[str, Dict[str, Set[int]]]

Read a user-provided SNPs CSV

Parameters

snp_csv
Path to SNPs CSV

Returns
Dictionary of contigs as keys and sets of genomic positions with SNPs as values

dynast.preprocessing.**read_snps**(*snps_path: str*) → Dict[str, Dict[str, Set[int]]]

Read SNPs CSV as a dictionary

Parameters

snps_path
Path to SNPs CSV

Returns
Dictionary of contigs as keys and sets of genomic positions with SNPs as values

4.2 Submodules

4.2.1 dynast.align

Module Contents

Functions

STAR_solo(fastqs: List[str], index_dir: str, out_dir: str, technology: dynast.technology.Technology, whitelist_path: Optional[str] = None, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', n_threads: int = 8, temp_dir: Optional[str] = None, nasc: bool = False, overrides: Optional[Dict[str, str]] = None) → Dict[str, str]	Align FASTQs with STARsolo.
align(fastqs: List[str], index_dir: str, out_dir: str, technology: dynast.technology.Technology, whitelist_path: Optional[str] = None, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', n_threads: int = 8, temp_dir: Optional[str] = None, nasc: bool = False, overrides: Optional[Dict[str, str]] = None)	Main interface for the *align* command.

dynast.align.**STAR_solo**(*fastqs: List[str]*, *index_dir: str*, *out_dir: str*, *technology: dynast.technology.Technology*, *whitelist_path: Optional[str] = None*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *nasc: bool = False*, *overrides: Optional[Dict[str, str]] = None*) → Dict[str, str]

Align FASTQs with STARsolo.

Parameters

fastqs
List of path to FASTQs. Order matters – STAR assumes the UMI and barcode are in read 2

index_dir
Path to directory containing STAR index

out_dir
Path to directory to place STAR output

technology
Technology specification

whitelist_path
Path to textfile containing barcode whitelist

strand
Strandedness of the sequencing protocol may be one of the following: *forward*, *reverse*, *unstranded*

n_threads
Number of threads to use

temp_dir
STAR temporary directory, defaults to *None*, which uses the system temporary directory

nasc
Whether or not to use STAR configuration used in NASC-seq pipeline

overrides
STAR command-line argument overrides

dynast.align.**align**(*fastqs: List[str]*, *index_dir: str*, *out_dir: str*, *technology: dynast.technology.Technology*, *whitelist_path: Optional[str] = None*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *nasc: bool = False*, *overrides: Optional[Dict[str, str]] = None*)

Main interface for the *align* command.

Parameters

fastqs
List of path to FASTQs. Order matters – STAR assumes the UMI and barcode are in read 2

index_dir
Path to directory containing STAR index

out_dir
Path to directory to place STAR output

technology
Technology specification

whitelist_path
Path to textfile containing barcode whitelist

strand
Strandedness of the sequencing protocol may be one of the following: *forward*, *reverse*, *unstranded*

n_threads
Number of threads to use

temp_dir
STAR temporary directory, defaults to *None*, which uses the system temporary directory

nasc
Whether or not to use STAR configuration used in NASC-seq pipeline

overrides
STAR command-line argument overrides

4.2.2 dynast.config

Module Contents

dynast.config.**PACKAGE_PATH**

dynast.config.**PLATFORM**

dynast.config.**BINS_DIR**

dynast.config.**MODELS_DIR**

dynast.config.**MODEL_PATH**

dynast.config.**MODEL_NAME = pi**

dynast.config.**RECOMMENDED_MEMORY**

dynast.config.**STAR_ARGUMENTS**

dynast.config.**STAR_SOLO_ARGUMENTS**

dynast.config.**NASC_ARGUMENTS**

dynast.config.**BAM_PEEK_READS = 500000**

dynast.config.**BAM_REQUIRED_TAGS = ['MD']**

dynast.config.**BAM_READGROUP_TAG = RG**

dynast.config.**BAM_BARCODE_TAG = CB**

dynast.config.**BAM_UMI_TAG = UB**

dynast.config.**BAM_GENE_TAG = GX**

dynast.config.**BAM_CONSENSUS_READ_COUNT_TAG = RN**

dynast.config.**COUNTS_SPLIT_THRESHOLD = 50000**

dynast.config.**VELOCITY_BLACKLIST = ['unassigned', 'ambiguous']**

4.2.3 dynast.consensus

Module Contents

Functions

consensus(bam_path: str, gtf_path: str, out_dir: str, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, quality: int = 27, add_RS_RI: bool = False, n_threads: int = 8, temp_dir: Optional[str] = None)	Main interface for the *consensus* command.

dynast.consensus.**consensus**(*bam_path: str*, *gtf_path: str*, *out_dir: str*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *quality: int = 27*, *add_RS_RI: bool = False*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*)

Main interface for the *consensus* command.

Parameters

bam_path
Path to BAM to call consensus from

gtf_path
Path to GTF used to build STAR index

out_dir
Path to output directory

strand
Strandedness of the protocol

umi_tag
BAM tag to use as UMIs

barcode_tag
BAM tag to use as cell barcodes

gene_tag
BAM tag to use as gene assignments

barcodes
Only consider these barcodes

quality
Quality threshold

add_RS_RI
Add RS and RI tags to BAM. Mostly useful for debugging.

n_threads
Number of threads to use

temp_dir
Temporary directory

4.2.4 dynast.constants

Module Contents

dynast.constants.**STATS_PREFIX = run_info**

dynast.constants.**STAR_SOLO_DIR = Solo.out**

dynast.constants.**STAR_GENE_DIR = Gene**

dynast.constants.**STAR_RAW_DIR = raw**

dynast.constants.**STAR_FILTERED_DIR = filtered**

dynast.constants.**STAR_VELOCYTO_DIR = Velocyto**

dynast.constants.**STAR_BAM_FILENAME = Aligned.sortedByCoord.out.bam**

dynast.constants.**STAR_BAI_FILENAME = Aligned.sortedByCoord.out.bai**

dynast.constants.**STAR_BARCODES_FILENAME = barcodes.tsv**

dynast.constants.**STAR_FEATURES_FILENAME = features.tsv**

dynast.constants.**STAR_MATRIX_FILENAME = matrix.mtx**

dynast.constants.**CONSENSUS_BAM_FILENAME = consensus.bam**

dynast.constants.**COUNT_DIR = count**

dynast.constants.**PARSE_DIR = 0_parse**

dynast.constants.**CONVS_FILENAME = convs.pkl.gz**

dynast.constants.**GENES_FILENAME = genes.pkl.gz**

dynast.constants.**CONVERSIONS_FILENAME = conversions.csv**

dynast.constants.**CONVERSIONS_INDEX_FILENAME = conversions.idx**

dynast.constants.**ALIGNMENTS_FILENAME = alignments.csv**

dynast.constants.**COVERAGE_FILENAME = coverage.csv**

dynast.constants.**COVERAGE_INDEX_FILENAME = coverage.idx**

dynast.constants.**SNPS_FILENAME = snps.csv**

dynast.constants.**COUNTS_PREFIX = counts**

dynast.constants.**ESTIMATE_DIR = estimate**

dynast.constants.**RATES_PREFIX = rates**

dynast.constants.**P_E_FILENAME = p_e.csv**

dynast.constants.**P_C_PREFIX = p_c**

dynast.constants.**AGGREGATE_FILENAME = aggregate.csv**

dynast.constants.**ADATA_FILENAME = adata.h5ad**

4.2.5 dynast.count

Module Contents

Functions

count(bam_path: str, gtf_path: str, out_dir: str, strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward', umi_tag: Optional[str] = None, barcode_tag: Optional[str] = None, gene_tag: str = 'GX', barcodes: Optional[List[str]] = None, control: bool = False, quality: int = 27, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})}), snp_threshold: Optional[float] = None, snp_min_coverage: int = 1, snp_csv: Optional[str] = None, n_threads: int = 8, temp_dir: Optional[str] = None, velocity: bool = True, strict_exon_overlap: bool = False, dedup_mode: typing_extensions.Literal[auto, exon, conversion] = 'auto', by_name: bool = False, nasc: bool = False, overwrite: bool = False)	Main interface for the *count* command.

dynast.count.**count**(*bam_path: str*, *gtf_path: str*, *out_dir: str*, *strand: typing_extensions.Literal[forward, reverse, unstranded] = 'forward'*, *umi_tag: Optional[str] = None*, *barcode_tag: Optional[str] = None*, *gene_tag: str = 'GX'*, *barcodes: Optional[List[str]] = None*, *control: bool = False*, *quality: int = 27*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*, *snp_threshold: Optional[float] = None*, *snp_min_coverage: int = 1*, *snp_csv: Optional[str] = None*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *velocity: bool = True*, *strict_exon_overlap: bool = False*, *dedup_mode: typing_extensions.Literal[auto, exon, conversion] = 'auto'*, *by_name: bool = False*, *nasc: bool = False*, *overwrite: bool = False*)

Main interface for the *count* command.

Parameters

bam_path
Path to BAM

gtf_path
Path to GTF

out_dir
Path to output directory

strand
Strandedness of technology

umi_tag
BAM tag to use as UMIs

barcode_tag
BAM tag to use as barcodes

gene_tag
BAM tag to use as genes

barcodes
List of barcodes to consider

control
Whether this is a control sample

quality
Quality threshold in detecting conversions

conversions
Set of conversions to quantify

snp_threshold
Call genomic locations that have greater than this proportion of specific conversions as a SNP

snp_min_coverage
Only consider genomic locations with at least this many mapping reads for SNP calling

snp_csv
CSV containing SNPs

n_threads
Number of threads to use

temp_dir
Temporary directory

velocity
Whether to quantify spliced/unspliced RNA

strict_exon_overlap
Whether spliced/unspliced RNA quantification is strict

dedup_mode
UMI deduplication mode

by_name
Whether to group counts by gene name instead of ID

nasc
Whether to match NASC-seq pipeline behavior

overwrite
Overwrite existing files

4.2.6 dynast.estimate

Module Contents

Functions

estimate(count_dirs: List[str], out_dir: str, reads: Union[typing_extensions.Literal[complete], List[typing_extensions.Literal[total, transcriptome, spliced, unspliced]]] = 'complete', barcodes: Optional[List[List[str]]] = None, groups: Optional[List[Dict[str, str]]] = None, ignore_groups_for_est: bool = True, genes: Optional[List[str]] = None, downsample: Optional[Union[int, float]] = None, downsample_mode: typing_extensions.Literal[uniform, cell, estimate.group] = 'uniform', cell_threshold: int = 1000, cell_gene_threshold: int = 16, control_p_e: Optional[float] = None, control: bool = False, method: typing_extensions.Literal[pi_g, alpha] = 'alpha', n_threads: int = 8, temp_dir: Optional[str] = None, nasc: bool = False, by_name: bool = False, seed: Optional[int] = None)	Main interface for the *estimate* command.

dynast.estimate.**estimate**(*count_dirs: List[str]*, *out_dir: str*, *reads: Union[typing_extensions.Literal[complete], List[typing_extensions.Literal[total, transcriptome, spliced, unspliced]]] = 'complete'*, *barcodes: Optional[List[List[str]]] = None*, *groups: Optional[List[Dict[str, str]]] = None*, *ignore_groups_for_est: bool = True*, *genes: Optional[List[str]] = None*, *downsample: Optional[Union[int, float]] = None*, *downsample_mode: typing_extensions.Literal[uniform, cell, estimate.group] = 'uniform'*, *cell_threshold: int = 1000*, *cell_gene_threshold: int = 16*, *control_p_e: Optional[float] = None*, *control: bool = False*, *method: typing_extensions.Literal[pi_g, alpha] = 'alpha'*, *n_threads: int = 8*, *temp_dir: Optional[str] = None*, *nasc: bool = False*, *by_name: bool = False*, *seed: Optional[int] = None*)

Main interface for the *estimate* command.

Parameters

count_dirs
Paths to directories containing *count* command output

out_dir
Output directory

reads
What read group(s) to quantify

barcodes
Cell barcodes

groups
Cell groups

ignore_groups_for_est
Ignore cell groups for final estimation

genes
Genes to consider

downsample
Downsample factor (float) or number (int)

donsample_mode
Downsampling mode

cell_threshold
Run estimation only for cells with at least this many counts

cell_gene_threshold
Run estimation for cell-genes with at least this many counts

control_p_e
Old RNA conversion rate (p_e), estimated from control samples

control
Whether this is a control sample

method
Estimation method to use

n_threads
Number of threads

temp_dir
Temporary directory

nasc
Whether to match NASC-seq pipeline behavior

by_name
Whether to group counts by gene name instead of ID

seed
Random seed

4.2.7 dynast.logging

Module Contents

dynast.logging.**logger**

4.2.8 dynast.main

Module Contents

Functions

print_technologies()	Displays a list of supported technologies along with whether a whitelist
setup_ref_args(parser: argparse.ArgumentParser, parent: argparse.ArgumentParser) → argparse.ArgumentParser	Helper function to set up a subparser for the *ref* command.
setup_align_args(parser: argparse.ArgumentParser, parent: argparse.ArgumentParser) → argparse.ArgumentParser	Helper function to set up a subparser for the *align* command.
setup_consensus_args(parser: argparse.ArgumentParser, parent: argparse.ArgumentParser) → argparse.ArgumentParser	Helper function to set up a subparser for the *consensus* command.
setup_count_args(parser: argparse.ArgumentParser, parent: argparse.ArgumentParser) → argparse.ArgumentParser	Helper function to set up a subparser for the *count* command.
setup_estimate_args(parser: argparse.ArgumentParser, parent: argparse.ArgumentParser) → argparse.ArgumentParser	Helper function to set up a subparser for the *estimate* command.
parse_ref(parser: argparse.ArgumentParser, args: argparse.Namespace, temp_dir: Optional[str] = None)	Parser for the *ref* command.
parse_align(parser, args, temp_dir=None)	
parse_consensus(parser: argparse.ArgumentParser, args: argparse.Namespace, temp_dir: Optional[str] = None)	Parser for the *consensus* command.
parse_count(parser: argparse.ArgumentParser, args: argparse.Namespace, temp_dir: Optional[str] = None)	Parser for the *count* command.
parse_estimate(parser: argparse.ArgumentParser, args: argparse.Namespace, temp_dir: Optional[str] = None)	Parser for the *estimate* command.
main()	

Attributes

COMMAND_TO_FUNCTION

dynast.main.**print_technologies**()

Displays a list of supported technologies along with whether a whitelist is provided for that technology.

dynast.main.**setup_ref_args**(*parser: argparse.ArgumentParser*, *parent: argparse.ArgumentParser*) → argparse.ArgumentParser

Helper function to set up a subparser for the *ref* command.

Parameters

parser
Argparse parser to add the *ref* command to

parent
Argparse parser parent of the newly added subcommand. Used to inherit shared commands/flags

Returns
The newly added parser

dynast.main.**setup_align_args**(*parser: argparse.ArgumentParser*, *parent: argparse.ArgumentParser*) → argparse.ArgumentParser

Helper function to set up a subparser for the *align* command.

Parameters

parser
Argparse parser to add the *align* command to

parent
Argparse parser parent of the newly added subcommand. Used to inherit shared commands/flags

Returns
The newly added parser

dynast.main.**setup_consensus_args**(*parser: argparse.ArgumentParser*, *parent: argparse.ArgumentParser*) → argparse.ArgumentParser

Helper function to set up a subparser for the *consensus* command.

Parameters

parser
Argparse parser to add the *consensus* command to

parent
Argparse parser parent of the newly added subcommand. Used to inherit shared commands/flags

Returns
The newly added parser

dynast.main.**setup_count_args**(*parser: argparse.ArgumentParser*, *parent: argparse.ArgumentParser*) → argparse.ArgumentParser

Helper function to set up a subparser for the *count* command.

Parameters

parser
Argparse parser to add the *count* command to

parent
Argparse parser parent of the newly added subcommand. Used to inherit shared commands/flags

Returns
The newly added parser

dynast.main.**setup_estimate_args**(*parser: argparse.ArgumentParser*, *parent: argparse.ArgumentParser*) → argparse.ArgumentParser

Helper function to set up a subparser for the *estimate* command.

Parameters

parser
Argparse parser to add the *estimate* command to

parent
Argparse parser parent of the newly added subcommand. Used to inherit shared commands/flags

Returns
The newly added parser

dynast.main.**parse_ref**(*parser: argparse.ArgumentParser*, *args: argparse.Namespace*, *temp_dir: Optional[str] = None*)

Parser for the *ref* command.

Parameters

parser
The parser

args
Command-line arguments dictionary, as parsed by argparse

temp_dir
Temporary directory

dynast.main.**parse_align**(*parser*, *args*, *temp_dir=None*)

dynast.main.**parse_consensus**(*parser: argparse.ArgumentParser*, *args: argparse.Namespace*, *temp_dir: Optional[str] = None*)

Parser for the *consensus* command.

Parameters

parser
The parser

args
Command-line arguments dictionary, as parsed by argparse

temp_dir
Temporary directory

dynast.main.**parse_count**(*parser: argparse.ArgumentParser*, *args: argparse.Namespace*, *temp_dir: Optional[str] = None*)

Parser for the *count* command.

Parameters

parser
The parser

args
Command-line arguments dictionary, as parsed by argparse

temp_dir
Temporary directory

dynast.main.**parse_estimate**(*parser: argparse.ArgumentParser*, *args: argparse.Namespace*, *temp_dir: Optional[str] = None*)

Parser for the *estimate* command.

Parameters

parser
The parser

args
Command-line arguments dictionary, as parsed by argparse

temp_dir
Temporary directory

dynast.main.**COMMAND_TO_FUNCTION**

dynast.main.**main**()

4.2.9 dynast.ref

Module Contents

Functions

STAR_genomeGenerate(fasta_path: str, gtf_path: str, index_dir: str, n_threads: int = 8, memory: int = 16 * 1024**3, temp_dir: Optional[str] = None) → Dict[str, str]	Generate a STAR index from a reference.
ref(fasta_path: str, gtf_path: str, index_dir: str, n_threads: int = 8, memory: int = 16 * 1024**3, temp_dir: Optional[str] = None)	Main interface for the `ref command.

dynast.ref.**STAR_genomeGenerate**(*fasta_path: str*, *gtf_path: str*, *index_dir: str*, *n_threads: int = 8*, *memory: int = 16 * 1024 ** 3*, *temp_dir: Optional[str] = None*) → Dict[str, str]

Generate a STAR index from a reference.

Parameters

fasta_path
Path to genome fasta

gtf_path
Path to GTF annotation

index_dir
Path to output STAR index

n_threads
Number of threads, defaults to *8*

memory
Suggested memory to use (this is not guaranteed), in bytes

temp_dir
Temporary directory

Retuurns:
Dictionary of generated index

dynast.ref.**ref**(*fasta_path: str*, *gtf_path: str*, *index_dir: str*, *n_threads: int = 8*, *memory: int = 16 * 1024 ** 3*, *temp_dir: Optional[str] = None*)

Main interface for the `ref command.

Parameters

fasta_path
Path to genome fasta

gtf_path
Path to GTF annotation

index_dir
Path to output STAR index

n_threads
Number of threads, defaults to *8*

memory
Suggested memory to use (this is not guaranteed), in bytes

temp_dir
Temporary directory

4.2.10 dynast.stats

Module Contents

Classes

Step	Class that represents a processing step.
Stats	Class used to collect run statistics.

class dynast.stats.**Step**(*skipped: bool = False*, ***kwargs*)

Class that represents a processing step.

start(*self*)

Signal the step has started.

end(*self*)

Signal the step has ended.

to_dict(*self*) → Dict[str, Any]

Convert this step to a dictionary.

class dynast.stats.**Stats**

Class used to collect run statistics.

start(*self*)

Start collecting statistics.

Sets start time, the command line call.

end(*self*)

End collecting statistics.

step(*self*, *key: str*, *skipped: bool = False*, ***kwargs*)

Register a processing step.

Any additional keyword arguments are passed to the constructor of *Step*.

save(*self*, *path: str*) → str

Save statistics as JSON to path.

to_dict(*self*) → Dict[str, Any]

Convert statistics to dictionary, so that it is easily parsed by the report-rendering functions.

4.2.11 dynast.technology

Module Contents

Functions

detect_strand(bam_path: str) → Optional[typing_extensions.Literal[forward, reverse, unstranded]]	Attempt to detect strandness by parsing the BAM header.

Attributes

Technology
BARCODE_UMI_TECHNOLOGIES
PLATE_TECHNOLOGIES
TECHNOLOGIES
TECHNOLOGIES_MAP
STRAND_MAP
BAM_STRAND_PARSER

dynast.technology.**Technology**

dynast.technology.**BARCODE_UMI_TECHNOLOGIES**

dynast.technology.**PLATE_TECHNOLOGIES**

dynast.technology.**TECHNOLOGIES**

dynast.technology.**TECHNOLOGIES_MAP**

dynast.technology.**STRAND_MAP**

dynast.technology.**BAM_STRAND_PARSER**

dynast.technology.**detect_strand**(*bam_path: str*) → Optional[typing_extensions.Literal[forward, reverse, unstranded]]

Attempt to detect strandness by parsing the BAM header.

Parameters

bam_path
Path to BAM

Returns
'unstranded', 'forward', or 'reverse if the strand was successfully detected. *None* otherwise.

4.2.12 dynast.utils

Module Contents

Classes

suppress_stdout_stderr	A context manager for doing a "deep suppression" of stdout and stderr in

Functions

get_STAR_binary_path() → str	Get the path to the platform-dependent STAR binary included with
get_STAR_version() → str	Get the provided STAR version.
combine_arguments(args: Dict[str, Any], additional: Dict[str, Any]) → Dict[str, Any]	Combine two dictionaries representing command-line arguments.
arguments_to_list(args: Dict[str, Any]) → List[Any]	Convert a dictionary of command-line arguments to a list.
get_file_descriptor_limit() → int	Get the current value for the maximum number of open file descriptors
get_max_file_descriptor_limit() → int	Get the maximum allowed value for the maximum number of open file
increase_file_descriptor_limit(limit: int)	Context manager that can be used to temporarily increase the maximum
get_available_memory() → int	Get total amount of available memory (total memory - used memory) in bytes.
make_pool_with_counter(n_threads: int) → Tuple[multiprocessing.Pool, multiprocessing.Value, multiprocessing.Lock]	Create a new Process pool with a shared progress counter.
display_progress_with_counter(counter: multiprocessing.Value, total: int, *async_results, desc: Optional[str] = None)	Display progress bar for displaying multiprocessing progress.
as_completed_with_progress(futures: Iterable[concurrent.futures.Future])	Wrapper around *concurrent.futures.as_completed* that displays a progress bar.
split_index(index: List[Tuple[int, int, int]], n: int = 8) → List[List[Tuple[int, int, int]]]	Split a conversions index, which is a list of tuples (file position,
downsample_counts(df_counts: pandas.DataFrame, proportion: Optional[float] = None, count: Optional[int] = None, seed: Optional[int] = None, group_by: Optional[List[str]] = None) → pandas.DataFrame	Downsample the given counts dataframe according to the `proportion` or
counts_to_matrix(df_counts: pandas.DataFrame, barcodes: List[str], features: List[str], barcode_column: str = 'barcode', feature_column: str = 'GX') → scipy.sparse.csr_matrix	Convert a counts dataframe to a sparse counts matrix.
split_counts(df_counts: pandas.DataFrame, barcodes: List[str], features: List[str], barcode_column: str = 'barcode', feature_column: str = 'GX', conversions: FrozenSet[str] = frozenset({'TC'})) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]	Split counts dataframe into two count matrices by a column.
split_matrix_pi(matrix: Union[numpy.ndarray, scipy.sparse.spmatrix], pis: Dict[Tuple[str, str], float], barcodes: List[str], features: List[str]) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]	Split the given matrix based on provided fraction of new RNA.
split_matrix_alpha(unlabeled_matrix: Union[numpy.ndarray, scipy.sparse.spmatrix], labeled_matrix: Union[numpy.ndarray, scipy.sparse.spmatrix], alphas: Dict[str, float], barcodes: List[str]) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]	Split the given matrix based on provided fraction of new RNA.
results_to_adata(df_counts: pandas.DataFrame, conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})}), gene_infos: Optional[dict] = None, pis: Optional[Dict[str, Dict[Tuple[str, Ellipsis], Dict[Tuple[str, str], float]]]] = None, alphas: Optional[Dict[str, Dict[Tuple[str, Ellipsis], Dict[str, float]]]] = None) → anndata.AnnData	Compile all results to a single anndata.

Attributes

run_executable
open_as_text
decompress_gzip
flatten_dict_values
mkstemp
all_exists
flatten_dictionary
flatten_iter
merge_dictionaries
write_pickle
read_pickle

dynast.utils.**run_executable**

dynast.utils.**open_as_text**

dynast.utils.**decompress_gzip**

dynast.utils.**flatten_dict_values**

dynast.utils.**mkstemp**

dynast.utils.**all_exists**

dynast.utils.**flatten_dictionary**

dynast.utils.**flatten_iter**

dynast.utils.**merge_dictionaries**

dynast.utils.**write_pickle**

dynast.utils.**read_pickle**

exception dynast.utils.**UnsupportedOSException**

Bases: Exception

Common base class for all non-exit exceptions.

class dynast.utils.**suppress_stdout_stderr**

A context manager for doing a "deep suppression" of stdout and stderr in Python, i.e. will suppress all print, even if the print originates in a compiled C/Fortran sub-function.

This will not suppress raised exceptions, since exceptions are printed

to stderr just before a script exits, and after the context manager has exited (at least, I think that is why it lets exceptions through). https://github.com/facebook/prophet/issues/223

__enter__(*self*)

__exit__(*self*, *_)

dynast.utils.**get_STAR_binary_path**() → str

Get the path to the platform-dependent STAR binary included with the installation.

Returns
Path to the binary

dynast.utils.**get_STAR_version**() → str

Get the provided STAR version.

Returns
Version string

dynast.utils.**combine_arguments**(*args: Dict[str, Any]*, *additional: Dict[str, Any]*) → Dict[str, Any]

Combine two dictionaries representing command-line arguments.

Any duplicate keys will be merged according to the following procedure: 1. If the value in both dictionaries are lists, the two lists are combined. 2. Otherwise, the value in the first dictionary is OVERWRITTEN.

Parameters

args
Original command-line arguments

additional
Additional command-line arguments

Returns
Combined command-line arguments

dynast.utils.**arguments_to_list**(*args: Dict[str, Any]*) → List[Any]

Convert a dictionary of command-line arguments to a list.

Parameters

args
Command-line arguments

Returns
List of command-line arguments

dynast.utils.**get_file_descriptor_limit**() → int

Get the current value for the maximum number of open file descriptors in a platform-dependent way.

Returns
The current value of the maximum number of open file descriptors.

dynast.utils.**get_max_file_descriptor_limit**() → int

Get the maximum allowed value for the maximum number of open file descriptors.

Note that for Windows, there is not an easy way to get this, as it requires reading from the registry. So, we just return the maximum for a vanilla Windows installation, which is 8192. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/setmaxstdio?view=vs-2019

Similarly, on MacOS, we return a hardcoded 10240.

Returns
Maximum allowed value for the maximum number of open file descriptors

dynast.utils.**increase_file_descriptor_limit**(*limit: int*)

Context manager that can be used to temporarily increase the maximum number of open file descriptors for the current process. The original value is restored when execution exits this function.

This is required when running STAR with many threads.

Parameters

limit
Maximum number of open file descriptors will be increased to this value for the duration of the context

dynast.utils.**get_available_memory**() → int

Get total amount of available memory (total memory - used memory) in bytes.

Returns
Available memory in bytes

dynast.utils.**make_pool_with_counter**(*n_threads: int*) → Tuple[multiprocessing.Pool, multiprocessing.Value, multiprocessing.Lock]

Create a new Process pool with a shared progress counter.

Parameters

n_threads
Number of processes

Returns
Tuple of (Process pool, progress counter, lock)

dynast.utils.**display_progress_with_counter**(*counter: multiprocessing.Value*, *total: int*, **async_results*, *desc: Optional[str] = None*)

Display progress bar for displaying multiprocessing progress.

Parameters

counter
Progress counter

total
Maximum number of units of processing

***async_results**
Multiprocessing results to monitor. These are used to determine when all processes are done.

desc
Progress bar description

dynast.utils.**as_completed_with_progress**(*futures: Iterable[concurrent.futures.Future]*)

Wrapper around *concurrent.futures.as_completed* that displays a progress bar.

Parameters

objects : *Iterator of concurrent.futures.Future*

dynast.utils.**split_index**(*index: List[Tuple[int, int, int]]*, *n: int = 8*) → List[List[Tuple[int, int, int]]]

Split a conversions index, which is a list of tuples (file position, number of lines, alignment position), one for each read, into *n* approximately equal parts. This function is used to split the conversions CSV for multiprocessing.

Parameters

index
index

n
Number of splits, defaults to *8*

Returns
List of parts

dynast.utils.**downsample_counts**(*df_counts: pandas.DataFrame*, *proportion: Optional[float] = None*, *count: Optional[int] = None*, *seed: Optional[int] = None*, *group_by: Optional[List[str]] = None*) → pandas.DataFrame

Downsample the given counts dataframe according to the `proportion` or `count` arguments. One of these two must be provided, but not both. The dataframe is assumed to be UMI-deduplicated.

Parameters

df_counts
Counts dataframe

proportion
Proportion of reads (UMIs) to keep

count
Absolute number of reads (UMIs) to keep

seed
Random seed

group_by
Columns in the counts dataframe to use to group entries. When this is provided, UMIs are no longer sampled at random, but instead grouped by this argument, and only groups that have more than `count` UMIs are downsampled.

Returns
Downsampled counts dataframe

dynast.utils.**counts_to_matrix**(*df_counts: pandas.DataFrame*, *barcodes: List[str]*, *features: List[str]*, *barcode_column: str = 'barcode'*, *feature_column: str = 'GX'*) → scipy.sparse.csr_matrix

Convert a counts dataframe to a sparse counts matrix.

Counts are assumed to be appropriately deduplicated.

Parameters

df_counts
Counts dataframe

barcodes
List of barcodes that will map to the rows

features
List of features (i.e. genes) that will map to the columns

barcode_column
Column in counts dataframe to use as barcodes, defaults to *barcode*

feature_column
Column in counts dataframe to use as features, defaults to *GX*

Returns
Sparse counts matrix

dynast.utils.**split_counts**(*df_counts: pandas.DataFrame*, *barcodes: List[str]*, *features: List[str]*, *barcode_column: str = 'barcode'*, *feature_column: str = 'GX'*, *conversions: FrozenSet[str] = frozenset({'TC'})*) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Split counts dataframe into two count matrices by a column.

Parameters

df_counts
Counts dataframe

barcodes
List of barcodes that will map to the rows

features
List of features (i.e. genes) that will map to the columns

barcode_column
Column in counts dataframe to use as barcodes

feature_column
Column in counts dataframe to use as features

conversions
Conversion(s) in question

Returns
count matrix of *conversion==0*, count matrix of *conversion>0*

dynast.utils.**split_matrix_pi**(*matrix: Union[numpy.ndarray, scipy.sparse.spmatrix]*, *pis: Dict[Tuple[str, str], float]*, *barcodes: List[str]*, *features: List[str]*) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Split the given matrix based on provided fraction of new RNA.

Parameters

matrix
Matrix to split

pis
Dictionary containing pi estimates

barcodes
All barcodes

features
All features (i.e. genes)

Returns
matrix of pis, matrix of unlabeled RNA, matrix of labeled RNA

dynast.utils.**split_matrix_alpha**(*unlabeled_matrix: Union[numpy.ndarray, scipy.sparse.spmatrix]*, *labeled_matrix: Union[numpy.ndarray, scipy.sparse.spmatrix]*, *alphas: Dict[str, float]*, *barcodes: List[str]*) → Tuple[scipy.sparse.csr_matrix, scipy.sparse.csr_matrix, scipy.sparse.csr_matrix]

Split the given matrix based on provided fraction of new RNA.

Parameters

unlabeled_matrix
unlabeled matrix

labeled_matrix
Labeled matrix

alphas
Dictionary containing alpha estimates

barcodes
All barcodes

features
All features (i.e. genes)

Returns
matrix of pis, matrix of unlabeled RNA, matrix of labeled RNA

dynast.utils.**results_to_adata**(*df_counts: pandas.DataFrame*, *conversions: FrozenSet[FrozenSet[str]] = frozenset({frozenset({'TC'})})*, *gene_infos: Optional[dict] = None*, *pis: Optional[Dict[str, Dict[Tuple[str, Ellipsis], Dict[Tuple[str, str], float]]]] = None*, *alphas: Optional[Dict[str, Dict[Tuple[str, Ellipsis], Dict[str, float]]]] = None*) → anndata.AnnData

Compile all results to a single anndata.

Parameters

df_counts
Counts dataframe, with complemented reverse strand bases

conversions
Conversion(s) in question

gene_infos
Dictionary containing gene information. If this is not provided, the function assumes gene names are already in the Counts dataframe.

pis
Dictionary of estimated pis

alphas
Dictionary of estimated alphas

Returns
Anndata containing all results

dynast.utils.**patch_mp_connection_bpo_17560**()

Apply PR-10305 / bpo-17560 connection send/receive max size update

See the original issue at https://bugs.python.org/issue17560 and https://github.com/python/cpython/pull/10305 for the pull request.

This only supports Python versions 3.3 - 3.7, this function does nothing for Python versions outside of that range.

Taken from https://stackoverflow.com/a/47776649

dynast.utils.**dict_to_matrix**(*d: Dict[Tuple[str, str], float]*, *rows: List[str]*, *columns: List[str]*) → scipy.sparse.csr_matrix

Convert a dictionary to a matrix.

Parameters

d
Dictionary to convert

rows
Row names

columns
Column names

Returns
A sparse matrix

4.3 Package Contents

dynast.**__version__ = 1.0.1**

CHAPTER

FIVE

REFERENCES

CHAPTER

SIX

NASC-SEQ

The new transcriptome alkylation-dependent scRNA-seq (NASC-seq) was developed by [Hendriks2019]. It uses Smart-seq, which is a plate-based scRNA-seq method that provides great read coverage, compared to droplet-based methods [Picelli2013]. Smart-seq experiments generate single or pairs of FASTQs for each cell sequenced, which dynast processes simultaneously.

- Sequencing technology: Smart-Seq2
- Induced conversion: T>C

6.1 Alignment

Here, we assume the appropriate STAR index has already been built (see *Building the STAR index with ref*). Since we have multiple sets of FASTQs, we need to prepare a FASTQ manifest CSV, instead of providing these as an argument to `dynast align`. The manifest CSV contains three columns where the first column is a unique cell name/ID, the second column is the path to the first FASTQ, and the third is the path to the second FASTQ. For single-end reads, the third column can be a single `-` character. Here is an example with two cells:

```
cell_1,path/to/R1.fastq.gz,path/to/R2.fastq.gz
cell_2,path/to/R1.fastq.gz,-
```

Then, we use this manifest as the input to `dynast align`.

```
dynast align -i path/to/STAR/index -o path/to/align/output -x smartseq manifest.csv
```

This will run STAR alignment and output files to `path/to/align/output`.

6.2 Quantification

The alignment BAM is generated at `path/to/align/output/Aligned.sortedByCoord.out.bam`, which we provde as input to `dynast count`. We also need to provide the gene annotation GTF that was used to generate the STAR index to `-g`.

```
dynast count -g path/to/GTF.gtf --barcode-tag RG path/to/align/output/Aligned.
↪sortedByCoord.out.bam -o path/to/count/output --conversion TC
```

This will quantify all RNA species and write the count matrices to `path/to/count/output/adata.h5ad`.

CHAPTER

SEVEN

SCSLAM-SEQ

scSLAM-seq was developed by [Erhard2019] and is the single-cell adaptation of thiol(SH)-linked alkylation for metabolic sequencing of RNA (SLAM-seq) [Herzog2017]. Similar to *NASC-seq*, scSLAM-seq is based on the Smart-seq protocol [Picelli2013]. Smart-seq experiments generate single or pairs of FASTQs for each cell sequenced, which dynast processes simultaneously.

- Sequencing technology: Smart-Seq2
- Induced conversion: T>C

7.1 Alignment

Here, we assume the appropriate STAR index has already been built (see *Building the STAR index with ref*). Since we have multiple sets of FASTQs, we need to prepare a FASTQ manifest CSV, instead of providing these as an argument to `dynast align`. The manifest CSV contains three columns where the first column is a unique cell name/ID, the second column is the path to the first FASTQ, and the third is the path to the second FASTQ. For single-end reads, the third column can be a single `-` character. Here is an example with two cells:

```
cell_1,path/to/R1.fastq.gz,path/to/R2.fastq.gz
cell_2,path/to/R1.fastq.gz,-
```

Then, we use this manifest as the input to `dynast align`.

```
dynast align -i path/to/STAR/index -o path/to/align/output -x smartseq --strand␣
↪unstranded manifest.csv
```

Note that we provide `--strand unstranded` because the Smart-seq protocol used with scSLAM-seq produces unstranded reads. This will run STAR alignment and output files to `path/to/align/output`.

7.2 Quantification

The alignment BAM is generated at `path/to/align/output/Aligned.sortedByCoord.out.bam`, which we provde as input to `dynast count`. We also need to provide the gene annotation GTF that was used to generate the STAR index to `-g`.

```
dynast count -g path/to/GTF.gtf --barcode-tag RG path/to/align/output/Aligned.
↪sortedByCoord.out.bam -o path/to/count/output --conversion TC --strand unstranded
```

Note that we provide `--strand unstranded` again because the Smart-seq protocol used with scSLAM-seq produces unstranded reads. This will quantify all RNA species and write the count matrices to `path/to/count/output/adata.h5ad`.

CHAPTER

EIGHT

SCNT-SEQ

The single-cell metabolically labeled new RNA tagging sequencing (scNT-seq) was developed by [Qiu2020]. It uses Drop-seq, which is a droplet-based scRNA-seq method [Macosko2015].

- Sequencing technology: Drop-seq
- Induced conversion: T>C

8.1 Alignment

Here, we assume the appropriate STAR index has already been built (see *Building the STAR index with ref*). A single sample will consist of a pair of FASTQs, one containing the cell barcode and UMI sequences and the other containing the biological cDNA sequences. Let's say these two FASTQs are `barcode_umi.fastq.gz` and `cdna.fastq.gz`.

```
dynast align -i path/to/STAR/index -o path/to/align/output -x dropseq cdna.fastq.gz␣
↪barcode_umi.fastq.gz
```

This will run STAR alignment and output files to `path/to/align/output`.

8.2 Consensus

Optionally, we can call consensus sequences for each UMI using `dynast consensus`. This command requires the alignment BAM and the gene annotation GTF that was used to generate the STAR index.

```
dynast consensus -g path/to/GTF.gtf --barcode-tag CB --umi-tag UB path/to/align/output/
↪Aligned.sortedByCoord.out.bam -o path/to/consensus/output
```

This will create a new BAM file named `path/to/consensus/output/consensus.bam`, which you can then use in the next step in place of the original alignment BAM.

8.3 Quantification

Finally, to quantify the number of labeled/unlabeled RNA, we run `dynast count` with the appropriate alignment BAM and the gene annotation GTF that was used to generate the STAR index to `-g`.

```
dynast count -g path/to/GTF.gtf --barcode-tag CB --umi-tag UB path/to/alignment.bam -o␣
↪path/to/count/output --conversion TC
```

where `path/to/alignment.bam` should be `path/to/align/output/Aligned.sortedByCoord.out.bam` if you did not run `dynast consensus`, or `path/to/consensus/output/consensus.bam` if you did.

This will quantify all RNA species and write the count matrices to `path/to/count/output/adata.h5ad`.

CHAPTER

NINE

SCI-FATE

The single-cell combinatorial indexing and messenger RNA labeling (sci-fate) was developed by [Cao2020].

- Sequencing technology: sci-fate
- Induced conversion: T>C

9.1 Alignment

Here, we assume the appropriate STAR index has already been built (see *Building the STAR index with ref*). A single sample will consist of a pair of FASTQs, one containing the cell barcode and UMI sequences and the other containing the biological cDNA sequences. Let's say these two FASTQs are `barcode_umi.fastq.gz` and `cdna.fastq.gz`.

```
dynast align -i path/to/STAR/index -o path/to/align/output -x scifate cdna.fastq.gz␣
↪barcode_umi.fastq.gz
```

This will run STAR alignment and output files to `path/to/align/output`.

9.2 Consensus

Optionally, we can call consensus sequences for each UMI using `dynast consensus`. This command requires the alignment BAM and the gene annotation GTF that was used to generate the STAR index.

```
dynast consensus -g path/to/GTF.gtf --barcode-tag CB --umi-tag UB path/to/align/output/
↪Aligned.sortedByCoord.out.bam -o path/to/consensus/output
```

This will create a new BAM file named `path/to/consensus/output/consensus.bam`, which you can then use in the next step in place of the original alignment BAM.

9.3 Quantification

Finally, to quantify the number of labeled/unlabeled RNA, we run `dynast count` with the appropriate alignment BAM and the gene annotation GTF that was used to generate the STAR index to `-g`.

```
dynast count -g path/to/GTF.gtf --barcode-tag CB --umi-tag UB path/to/alignment.bam -o␣
↪path/to/count/output --conversion TC
```

where `path/to/alignment.bam` should be `path/to/align/output/Aligned.sortedByCoord.out.bam` if you did not run `dynast consensus`, or `path/to/consensus/output/consensus.bam` if you did.

This will quantify all RNA species and write the count matrices to `path/to/count/output/adata.h5ad`.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Dobin2013] https://doi.org/10.1093/bioinformatics/bts635

[Picelli2013] https://doi.org/10.1038/nmeth.2639

[Macosko2015] https://doi.org/10.1016/j.cell.2015.05.002

[Herzog2017] https://doi.org/10.1038/nmeth.4435

[Jürges2018] https://doi.org/10.1093/bioinformatics/bty256

[Erhard2019] https://doi.org/10.1038/s41586-019-1369-y

[Hendriks2019] https://doi.org/10.1038/s41467-019-11028-9

[Cao2020] https://doi.org/10.1038/s41587-020-0480-9

[Qiu2020] https://doi.org/10.1038/s41592-020-0935-4

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W