alevin-fry

Release 0.7.0

Dongze He, Mohsen Zakeri, Hirak Sarkar, Charlotte Soneson, Avi S

Jun 29, 2023

CONTENTS:

CHAPTER

ONE

WHAT IS ALEVIN-FRY?

Alevin-fry is a program exposing a suite of tools for single-cell sequencing analysis. It makes use of the selective-alignment and tag processing framework of alevin.

1.1 Overview

alevin-fry ` is a suite of tools for the rapid, accurate and memory-frugal processing single-cell and single-nucleus sequencing data. It consumes RAD files generated by *salmon alevin*, and performs common operations like generating permit lists, and estimating the number of distinct molecules from each gene within each cell. The focus in *alevin-fry* ` is on safety, accuracy and efficiency (in terms of both time and memory usage).

You can read the paper describing alevin fry, "Alevin-fry unlocks rapid, accurate, and memory-frugal quantification of single-cell RNA-seq data" [here](https://www.nature.com/articles/s41592-022-01408-3), and the pre-print [on bioRxiv](https://www.biorxiv.org/content/10.1101/2021.06.29.450377v1).

1.2 Other resources for alevin-fry

In addition to the current documentation page, there are numerous other resources to help you learn more about alevin-fry, how to process data using this program, and how to further process the output of alevin-fry in downstream analysis.

1.2.1 Tutorials

A collection of tutorials describing how to process different types of data with *alevin-fry* and describing different features of *alevin-fry* is available here.

1.2.2 FAQ

We hope to make use of GitHub discussions to answer frequently asked questions, and to discuss other issues relevant to the development and use of *alevin-fry*. You can visit the GitHub discussion page for alevin-fry here. GitHub discussions are also a good place to raise large-scale feature requests to see if they make sense in the context of *alevin-fry*. For small-scale feature requests, or to report bugs or unexpected behavior you encounter when processing data with *alevin-fry*, please make use of our GitHub issues page.

1.2.3 Quality Control

The alevinQC package supports quality control of *alevin-fry* processed data.

1.2.4 Easy loading of USA-mode data

The fishpond package contains many methods for making the ingestion of quantification results generated by salmon and *alevin-fry* into R easy. In particular, you can find documentation on the loadFry function here. This makes it easy to import USA-mode quantification results into a SingleCellExperiment object, and to properly extract or combine the spliced, unspliced, and ambiguous count components.

1.3 Installing alevin-fry

Alevin-fry can be installed using a package manager such as `conda`, or built from source.

1.3.1 Installing with bioconda

Alevin-fry is available for both x86 linux and OSX platforms using bioconda.

With `bioconda` in the appropriate place in your channel list, you should simply be able to install via:

```
$ conda install alevin-fry
```

1.3.2 Installing from source

If you want to use features or fixes that may only be available in the latest develop branch (or want to build for a different architecture), then you have to build from source. Luckily, `cargo` makes that easy; see below.

Alevin-fry is built and tested with the latest (major & minor) stable version of Rust. While it will likely compile fine with older versions of Rust, this is not a guarantee and is not a support priority. Unlike with C++, Rust has a frequent and stable release cadence, is designed to be installed and updated from user space, and is easy to keep up to date with rustup. Thanks to cargo, building should be as easy as:

```
$ cargo build --release
```

subsequent you will want to place `alevin-fry` in your `PATH`. This can be done (in bash-like shells) using:

```
$ export PATH=`pwd`/target/release/:$PATH
```

To ensure that `alevin-fry` remains in your path between logins, you should make sure the path to `target/release/` shown above is set in the `PATH` variable in the appropriate file for your shell (e.g. in `~/.profile`, `~/.bashrc` etc.).

1.4 Getting Started

1.4.1 Running the alevin-fry pipeline

First, we need to generate a RAD file using alevin. The RAD file is created by mapping the sequencing reads against an index of the reference. We recommend using a splici reference index. The mappings can be generated using either selective-alignment or pseudoalignment against the transcriptome (with the `--rad` or `--sketch` flags, respectively). **Note**, however, that alevin-fry does not currently support RAD files aligned against a decoy-aware index, so that indices used for RAD file generation should be prepared without decoy sequnece. For a chromium v2 set of read files, the command would look like the following:

```
$ salmon alevin -lISR --chromium -1 <read1_files> -2 <read2_files> -o <alevin_odir> -i
↪<index> -p <num_threads> --sketch
```

Given the output directory generated above, the next step is to let alevin-fry generate the permit list. First, we grab the 10x Chromium version 2 permit list (if we had Chromium v3 chemistry, we would use that permit-list instead):

```
$ wget https://umd.box.com/shared/static/jbs2wszgbj7k4ic2hass9ts6nhqkwq1p -O 10x_v2_
↪permit.txt
```

Now, we can use this permit list to scan the cell barcodes actually encountered in our reads and determine a set of cells that were likely present in our sample:

```
$ alevin-fry generate-permit-list --input <alevin_odir> --expected-ori fw --output-dir
↪<fry_odir> --unfiltered-pl 10x_v2_permit.txt
```

Next, given the permit list and barcode mapping (which resides in the *<fry_odir>* directory), we collate the original RAD file using the command below.

```
$ alevin-fry collate -i <fry_odir> -r <alevin_odir> -t <num_threads>
```

Finally, we quantify the collated rad file using the *cr-like* resolution strategy using the *quant* command below.

```
$ alevin-fry quant -i <fry_odir> -m <tg_map> -t <num_threads> -r cr-like -o <fry_odir>
```

Note that with the exception of the *generate-permit-list* command, the other *alevin-fry* commands are designed to scale well with the number of provided threads. Thus, if you have multiple threads to use, then you can provide the appropriate argument to the *-t* option.

1.4.2 Detailed information on the alevin-fry commands

There are a (growing) number of different sub-commands for `alevin-fry`. To learn more about the different commands an their options check the *commands* section of the documentation.

1.5 alevin-fry commands

The `alevin-fry` program exposes a number of different commands that are responsible for exposing different capabilities and performing different steps of the processing pipeline. The list of current and valid are documented below.

1.5.1 generate-permit-list

This command takes as input an output directory containing a RAD file (created by running alevin with the `--rad` and/or `--sketch` flags), and it determines what cell barcodes should be associated with "true" cells, which should be corrected to some "true" barcode, and which should simply be ignored / discarded. This command has 4 required arguments; the path to an input directory `--input`, the path to an output directory `--output-dir` (which will be created if it doesn't exist), the expected orientation of properly mapped reads `--expected-ori` (the options are 'fw' (filters out alignments to the reverse complement strand), 'rc' (filter out alignments to the forward strand) and 'both' or 'either' (do not filter any alignments)), and then one of the following mutually exclusive options (which determines how the "true" barcodes are decided):

- `--knee-distance`: This flag will use the distance method that is used in the whitelist command of UMI-tools to attempt to automatically determine the number of true barcodes. Briefly, this method first counts the number of reads associated with each barcode, and then sorts the barcodes in descending order by their associated read count. It then constructs the cumulative distribution function from this sorted list of frequencies. Finally, it applies an iterative algorithm to attempt to determine the optimal number of barcodes to include by looking for a "knee" or "elbow" in the CDF graph. The algorithm considers each barcode in the CDF where it's x-coordinate is equal to this barcode's rank divided by the total number of barcodes (i.e. its normalized rank) and the y-coordinate is equal to the (normalized) cumulative frequency achieved at this barcode. It then computes the distance of this barcode from the line x=y (defined by the start and end of the CDF). The initial knee is predicted as the point that has the maximum distance from the x=y line. The algorithm is iterative, because experiments with many low-quality barcodes may predict too many valid barcodes using this method. Thus, the algorithm is run repeatedly, each time considering a prefix of the CDF from index 0 through the previous knee's index * 5. Once two subsequent iterations of the algorithm return the same knee point, the algorithm terminates.
- `--force-cells <ncells>`: This option will count the number of reads associated with each barcode, and sort the barcodes in descending order of frequency. Then, it will consider the first <ncells> barcodes to be valid. Any barcode that has a number of reads >= to the <ncells>-th barcode will be considered part of the permit list, all others will not (but will be considered for correction to this permit list).
- `--valid-bc <bcfile>`: This option will read the provided file <bcfile> and treat it as an explicitly-provided list of true, filtered barcodes (i.e. a list of barcodes believed to belong to a set of high-confidence cells truly present in the given sample). Barcodes appearing in this list will be considered to correspond to true and filtered cells, and barcodes will be corrected to this list. This flag is *not* designed to perform unfiltered quantification (i.e. correcting to a list of all *possible* barcodes generated by a technology, like e.g. the 10x v3 permit list). To correct against an *unfiltered* permit list, you should use the `--unfiltered-pl` flag described below.
- `--unfiltered-pl <plist>`: This option accepts as an argument a list of *possible* barcodes for the sample. For example, this is the flag you should use if you wish to provide an "external permit list", like the 10x v2 or 10x v3 permit lists. Unilike with the `--valid-bc` flag, the list passed to this argument is the set of all possible barcodes for the technology being processed, and it is likely that most of the barcodes in the file may not correspond to cells present in this particular sample. When using this argument, you may also pass the `--min-reads` argument to determine the minimum frequency with which a barcode must be seen in order to be retained. The algorithm used here will pass over the input records (mapped reads) and count how many times each of the barcodes in the unfiltered permit list occur exactly. Any barcode ocurring `>=` `min-reads` times will be considered as a present cell. Subsequently, all barcodes that did not match a present cell will be searched (at an edit distance of up to 1) againt the barcodes determined to correspond to present cells. If an initially non-matching barcode has a unique neighbor among the barcodes for present cells, it will be corrected to that barcode, but if it has no 1-edit neighbor,

or if it has 2 or more 1-edit neighbors among that list (i.e. it's correction would be ambiguous), then the record is discarded.

- `--min-reads <threshold>`: This flag is meant to be used (and currently only applied) in conjunction with `--unfiltered-pl`. Any barcodes from the provided permit list that have `>= <threshold>` exact occurrences in the input file will be deemed as present cells and will be passed on to subsequent phases of quantification. Barcodes occurring `< threshold` number of times will be corrected against the set of present cells using the procedure described above.
- `--expect-cells <ncells>`: This option uses the provided <ncells> as a hint, and tries to choose a robust cutoff around this value. The functionality of this option corresponds, approximately to what you would get from passing the flag `--soloCellFilter <ncells> 0.99 10` to STARsolo.

output

The `generate-permit-list` command outputs a number of different files in the output directory. Not all files are relevant to users of `alevin-fry`, but the files are described here.

1. The file `all_freq.bin` is a binary file that records, for each distinct barcode in the input RAD file, the number of read records that were tagged with this barcode.
2. The file `permit_freq.bin` is a binary file that lists, for each barcode in the input RAD file that is determined to be a *true* barcode, the number of read records associated with this barcode.
3. The file `permit_map.bin` is a binary file (a serde serialized HashMap) that maps each barcode in the input RAD file that is within an edit distance of 1 to some *true* barcode to the barcode to which it corrects. This allows the `collate` command to group together all of the read records corresponding to the same *corrected* barcode.
4. The file `generate_permit_list.json` that is a JSON file containing information about the run of the command (currently, just the expected orientation).

1.5.2 collate

This command takes as input a directory containing a RAD file (created by running alevin with the `--justAlign` and/or `--sketch` flags), as well as the directory generated as the result of running the `generate-permit-list` command of `alevin-fry`, and it will produce an output RAD file that is *collated* by (corrected) cellular barcode. The collated RAD file can then be quantified with the `alevin-fry quant` command. It also takes two other arguments (described below) that dictate how the collation and filtering will be performed.

- `-r, --rad-dir <rad-dir>` : The directory containing the RAD file to be collated. This is the *same* directory on which you have previously run `generate-permit-list` and that was obtained by running `alevin` with the `--justAlign` flag).
- `-i, --input-dir <input-dir>` : The input directory. This is the directory that was the *output* of `generate-permit-list`. This directory contains information computed by the `generate-permit-list` command that will allow successful collation and barcode correction. This is also the directory where the collated RAD file will be *output*.
- `--compress` : This optional flag will tell `alevin-fry` to compress the output collated RAD file. The file will be compressed using the Snappy compression format (via the excellent snap crate. If this option is passed, the output file will be written to `map.collated.rad.sz` rather than `map.collated.rad`, and the corresponding status of the file's compression will be written to `collate.json` in the output file. *Note*: The choice to use compression or not has no effect on the final result or the correctness of the output, but it may have some moderate performance implications. Specifically, it is potentially worth using this flag if you want to minimize disk space, and if you are using a sufficiently large number of threads (as compression happens in parallel, a sufficient number of threads will allow the compressed RAD file to be generated as quickly as the uncompressed). However, because some internal buffers must be duplicated during parallel compression, the collate step can use a bit more memory if run

with the `--compress` flag, though the memory usage should still be small and stable over different sized inputs. There can also be an effect on quantification speed (since the collated RAD file will be decompressed on the fly during quantification), but it should be small since Snappy decompresses very fast, and decompression will only be the limiting factor if you are using a simple resolution strategy (e.g. naive or cr-like) and many quantification threads.

- `-m, --max-records <max-records>` : The maximum number of read records to keep in memory at once during collation. The `collate` command will pass over the input RAD file multiple times collecting the records associated with a set of (corrected) cellular barcodes so that they can be written out in collated format to the output RAD file. This parameter determines (approximately) how many records will be held in memory at once, and therefore determines the memory usage of the `collate` command. The larger the value used the faster the collation process will be, since fewer passes are made. The smaller this value, the lower the memory usage will be, at the cost of more passes. The default value is 30,000,000. Note that this determines the number of records *approximately*, because a specific barcode will never be split across multiple collation passes. The algorithm employed is to collect the reads associated with different cellular barcodes in the current pass until the number of reads to be collected *first exceeds* this value.

output

The `collate` command will output all files it creates in the expected format in the output directory that is specified. It will write a file name `map.collated.rad` (or `map.collated.rad.sz` if run with the `--compress` flag), one named `unmapped_bc_count_collated.bin`, and one named `collate.json` in the directory specified by `-i`.

1.5.3 quant

The `quant` command takes a collated RAD file and performs feature (e.g. gene) quantification, outputting a sparse matrix of de-duplicated counts as well as a list of labels for the rows and columns. The `quant` command takes an input directory containing the collated RAD file, a transcript-to-gene map, an output directory where the results will be written, and a "resolution strategy" (described below). Quantification is multi-threaded, so it also, optionally, takes as an arguments the number of threads to use concurrently.

The transcript-to-gene map (provided using the `-m` or `--tg-map` option) should be either:

1. A three-column (headerless) tab-separated file where the first column contains a target name, the second column contains the corresponding gene feature to which this target belongs and the third column contains either `S` or `U`, with `S` denoting the corresponding feature should be attributed to the spliced status of its parent gene and `U` denoting that it should be attributed to the unspliced status of its parent gene.
2. A two-column (headerless) tab-separated file where the first column contains a transcript name and the second column contains the corresponding gene name for this transcript.

The `quant` command exposes a number of different resolution strategies. Note: If you are providing a three-column transcript-to-gene map, and hence quantifying in Unspliced/Spliced/Ambiguous (USA) mode, then only the `cr-like` and `cr-like-em` resolution modes are currently available. The different UMI resolution strategies are:

- `cr-like` : This strategy is like the one adopted in cell-ranger, except that it does not first collapse 1-edit-distance UMIs. Within each cell barcode, a list of (gene, UMI, count) tuples is created. If a read maps to more than one gene, then it generates more than one such tuple. The tuples are then sorted lexicographically (first by gene id, then by UMI, and then by count). Any UMI that aligns to only a single gene is assigned to that gene. UMIs that align to more than one gene are assigned to the gene with the highest count for this UMI. If there is a tie for the highest count gene for this UMI, then the corresponding reads are simply discarded.
- `cr-like-em` : This strategy is like `cr-like`, except that when a UMI has genes to which it matches with equal frequency, rather than discard the UMIs, the genes are treated as an equivalence class, and the counts for each gene are determined via an expectation maximization algorithm.

- `parsimony-em/full` : This implements the algorithm described in the alevin paper. Briefly, it builds a graph among the set of reads that align to an overlapping set of transcripts and that have similar (within an edit distance of 1) UMIs. It then attempts to find a parsimonious cover for this graph using the fewest number of possible transcripts. If a unique parsimonious cover is found, then the (deduplicated) reads are assigned directly to the genes that yield the most parsimonious cover. If multiple equally-parsimonious covers exist, then the reads are considered multi-mapping at the gene level and they are probabilistically resolved using an expectation maximization (EM) algorithm.
- `parsimony` : This strategy is the same as "full", except that it does *not* probabilistically resolve reads that remain as gene-multimapping after applying the parsimony criterion. Instead, reads that do not have a unique most-parsimonious assignment are discarded.
- `parsimony-gene` : This strategy is the same as `parsimony` above, except that mappings of UMIs to transcripts are projected to their corresponding gene *before* the relevant graph (Parsimonious UMI Graph) is constructed. Thus, the vertices of the graph consist of sets of gene labels rather than sets of transcript labels. This method may be less precise than the `parsimony` method (i.e. may wrongly group together UMIs that arise from different transcripts within the same gene), but it simultaneously likely to be more robust to misannotation or incomplete annotation (e.g. incomplete UTR annotation).
- `parsimony-gene-em` : This strategy is the same as `parsimony-above` above, except that, like `parsimony-em` any graphs that exhibit a multi-gene cover will have the multimapping resolved probabilistically with an EM algorithm..
- `trivial` : This strategy does not search for 1 edit-distance neighbors of UMIs. Instead, it first discards any reads that multi-map at the gene level. The reads that remain then all map uniquely to a single gene. These reads are deduplicated by (exact) UMI, and the number of distinct UMIs mapping to each gene are taken as that gene's count in the current cell. **Note**: This resolution strategy is not available in USA mode.

Additionally, this command can optionally take the following flags (note that not all resolution strategies are compatible with these flags):

- `-d, --dump-eqclasses` : This flag will cause a gene-level, UMI-deduplicated, equivalence class counts file to be written to the output directory in addition to the gene-level count matrix. This can be used for subsequent analyses where gene-ambiguous reads have been neither resovled nor discarded.
- `-b, --num-bootstraps` : This flag will cause bootstrap inferential replicate information to be written to the output directory. This provides a measure of the inferential uncertainty in the gene-level estimates provided by `alevin-fry` when run with a method using the EM algorithm for gene-level abundance estimation. This information can be used with downstream testing, like differential expression testing using swish. This flag is only meaningful with the `cr-like-em` or `full` resolution modes.
- `--summary-stat` : This flag will write the summary statistics of the bootstrap replicates (i.e. the mean and variance of the inferential replicates). This provides the most important information for uncertainty-aware downstream analysis, while requiring much less storage space than the full bootstrap replicate information. This flag is only meaningful when `--num-bootstraps` is meaningful.
- `--quant-subset <SFILE>` : This optional argument provides a file containing list of barcodes to quantify (one barcode per line, written as a string), those not in this list will be ignored during inference and will not appear in the output quantification matrix. If this argument is not provided, then all of the original barcodes will be quantified.
- `--use-mtx` : This flag will cause the output to be written in matrix market coordinate format (which is the default).
- `--use-eds` : This flag will cause the output to be written in EDS format rather than in matrix market format.

There are also a few flags that are not immediately exposed:

- `--umi-edit-dist <EDIST>` : This option takes a parameter that sets the Hamming distance within which potentially colliding UMIs will be considered for correction. With resolution modes `parsimony`, `parsimony-em`, parsimony-gene or parsimony-gene-em the valid values are 0 and 1 (and the default is 1). With other resolution modes, the default (and currently the only supported value) is 0.

- --large-graph-thresh <NVERT> : This option takes a parameter that sets the order (number of nodes) of a PUG above which an alternative (faster) resolution strategy will be applied. This option only has an effect for parsimony, parsimony-em, parsimony-gene or parsimony-gene-em resolution modes. The default value is 1000.

output

The output of the quant command consists of 5 files: quants_mat_rows.txt, quants_mat.mtx (or counts.eds.gz if run with the --use-eds flag), quants_mat_cols.txt, quant.json, and featureDump.txt. The quant.json file contains information about the quantification run, such as the method used for UMI resolution. The featureDump.txt file contains cell-level information designed to be useful in post-quantification cell filtering (better determining "true" cells from background, noise, doublets etc.). The other three files all correspond to quantification information.

If quant was executed in USA mode, then the resulting count matrix will be of dimension C``x``3G where C is the number of quantified cells (barcodes) and G is the number of genes. This is because, in USA mode, alevin-fry quantifies the UMI count attributable to each splicing state of each gene in each cell, where the splicing state is one of spliced (S), unspliced (U) or ambiguous (A). If quant was run with a two-column transcript-to-gene map (not in USA-mode), then the resulting count matrix will be a C``x``G matrix, as splicing status is not tracked. For more details on USA mode and its uses, please read the alevin-fry paper or preprint, or the corresponding tutorial.

The quants_mat.mtx is a matrix market coordinate format file (or if running with --use-eds then counts.eds.gz is a gzipped file in EDS format) that stores the gene-by-cell expression matrix. The two other files provide the labels for the rows and columns of this matrix. The quants_mat_cols.txt file is a text file that contains the names of the rows of the matrix, in the order in which it is written, with one gene name written per line. The quants_mat_rows.txt file is a text file that contains the names of the columns of the matrix, in the order in which it is written, with one barcode name written per line.

1.5.4 infer

The infer command takes a gene-resolved equivalence class count matrix, along with the equivalence class description file (both output when running quant with the -d flag), and performs inference via the EM to reduce the matrix to a cell-by-gene level count matrix.

This functionality makes it possible to separate the UMI resolution step (which may result in UMIs being assigned to equivalence classes of genes rather than individual genes), and the gene-level estimation step, that attempts to resolve gene-multimapping UMIs.

This command can takes the following options :

- -c, --count-mat <eqc-mat> : This provides the path to the (mtx format) matrix of cells by equivalence class counts. **Note**: It is assumed that the parent directory where eqc-mat is located will also contain a file called quants_mat_rows.txt containing the row names of the matrix and a file called quants_mat_cols.txt containing the column names of the files. The infer command will not run if these other input files are absent from the parent directory of eqc-mat.
- -e, --eq-labels <eq-labels> : This provides the path to the file containing the gene labels of the equivalence class description.
- -o, --output-dir <output-dir> : This provides the output file directory the quantification matrix, barcodes (row names), and genes (column names) will be written.
- --quant-subset <sfile> : This optional argument provides a file containing list of barcodes to quantify (one barcode per line, written as a string), those not in this list will be ignored during inference and will not appear

in the output quantification matrix. If this argument is not provided, then all of the original barcodes will be quantified.

- `-t, --threads <threads>` : This option provides the number of threads to use for processing [default: number of hardware threads].

output

The output of the `infer` command is written in the provided `output-dir`. It consists of the cell-by-gene count matrix derived from the input cell-by-equivalence-class count matrix, as well as a `quants_mat_rows.txt` and `quants_mat_cols.txt` file providing the row and column names for the output matrix, respectively.

1.6 License

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search