Salmon Documentation

Release 0.13.1

Rob Patro, Geet Duggal, Mike Love, Rafael Irizarry and Carl Kings

May 21, 2019

Contents

Contents:

CHAPTER 1

Requirements

1.1 Binary Releases

Pre-compiled binaries of the latest release of Salmon for a number different platforms are available available under the Releases tab of Salmon's GitHub repository. You should be able to get started quickly by finding a binary from the list that is compatible with your platform. Additionally, you can obtain a Docker image of the latest version from DockerHub using:

```
> docker pull combinelab/salmon
```

1.2 Requirements for Building from Source

- A C++11 conformant compiler (currently tested with GCC>=4.7 and Clang>=3.4)
- CMake. Salmon uses the CMake build system to check, fetch and install dependencies, and to compile and install Salmon. CMake is available for all major platforms (though Salmon is currently unsupported on Windows.)

CHAPTER 2

Installation

After downloading the Salmon source distribution and unpacking it, change into the top-level directory:

```
> cd salmon
```

Then, create and out-of-source build directory and change into it:

```
> mkdir build
> cd build
```

Salmon makes extensive use of Boost. We recommend installing the most recent version (1.55) systemwide if possible. If Boost is not installed on your system, the build process will fetch, compile and install it locally. However, if you already have a recent version of Boost available on your system, it make sense to tell the build system to use that.

If you have Boost installed you can tell CMake where to look for it. Likewise, if you already have Intel's Threading Building Blocks library installed, you can tell CMake where it is as well. The flags for CMake are as follows:

- -DFETCH_BOOST=TRUE – If you don't have Boost installed (or have an older version of it), you can provide the FETCH_BOOST flag instead of the BOOST_ROOT variable, which will cause CMake to fetch and build Boost locally.
- -DBOOST_ROOT=<boostdir> – Tells CMake where an existing installtion of Boost resides, and looks for the appropritate version in <boostdir>. This is the top-level directory where Boost is installed (e.g. /opt/local).
- -DTBB_INSTALL_DIR=<tbbroot> – Tells CMake where an existing installation of Intel's TBB is installed (<tbbroot>), and looks for the apropriate headers and libraries there. This is the top-level directory where TBB is installed (e.g. /opt/local).
- -DCMAKE_INSTALL_PREFIX=<install_dir> – <install_dir> is the directory to which you wish Salmon to be installed. If you don't specify this option, it will be installed locally in the top-level directory (i.e. the directory directly above "build").

There are a number of other libraries upon which Salmon depends, but CMake should fetch these for you automatically.

Setting the appropriate flags, you can then run the CMake configure step as follows:

```
> cmake [FLAGS] ..
```

The above command is the cmake configuration step, which *should* complain if anything goes wrong. Next, you have to run the build step. Depending on what libraries need to be fetched and installed, this could take a while (specifically if the installation needs to install Boost). To start the build, just run make.

```
> make
```

If the build is successful, the appropriate executables and libraries should be created. There are two points to note about the build process. First, if the build system is downloading and compiling boost, you may see a large number of warnings during compilation; these are normal. Second, note that CMake has colored output by default, and the steps which create or link libraries are printed in red. This is the color chosen by CMake for linking messages, and does not denote an error in the build process.

Finally, after everything is built, the libraries and executable can be installed with:

```
> make install
```

You can test the installation by running

```
> make test
```

This should run a simple test and tell you if it succeeded or not.

CHAPTER 3

Salmon

Salmon is a tool for **wicked-fast** transcript quantification from RNA-seq data. It requires a set of target transcripts (either from a reference or *de-novo* assembly) to quantify. All you need to run Salmon is a FASTA file containing your reference transcripts and a (set of) FASTA/FASTQ file(s) containing your reads. Optionally, Salmon can make use of pre-computed alignments (in the form of a SAM/BAM file) to the transcripts rather than the raw reads.

The **mapping**-based mode of Salmon runs in two phases; indexing and quantification. The indexing step is independent of the reads, and only need to be run one for a particular set of reference transcripts. The quantification step, obviously, is specific to the set of RNA-seq reads and is thus run more frequently. For a more complete description of all available options in Salmon, see below.

Note: Mapping validation in mapping-based mode

Mapping validation, enabled by the `--validateMappings` flag, is a major feature enhancement introduced in recent versions of salmon. When salmon is run with mapping validation, it adopts a considerably more sensitive scheme that we have developed for finding the potential mapping loci of a read, and score potential mapping loci using the chaining algorithm introdcued in minimap2[5]. Finally, it scores and validates these mappings using the score-only, SIMD, dynamic programming algorithm of ksw2[6]. The use of mapping validation implies the use of range factorization, as mapping scores become very meaningful with this option. Mapping validation can improve the accuracy, sometimes considerably, over the faster, but less-precise default mapping algorithm. As of salmon v0.13.1, we highly recommend all users adopt mapping validation unless they have a specific reason to avoid it. It is likely that this option will be enabled by default in a future release. Also, there are a number of options and flags that allow the user to control details about how the scoring is carried out, including setting match, mismatch, and gap scores, and choosing the minimum score below which an alignment will be considered invalid, and therefore not used for the purposes of quantification.

The **alignment**-based mode of Salmon does not require indexing. Rather, you can simply provide Salmon with a FASTA file of the transcripts and a SAM/BAM file containing the alignments you wish to use for quantification.

[5] Li, Heng. "Minimap2: pairwise alignment for nucleotide sequences." Bioinformatics 34.18 (2018): 3094-3100.
[6] Global alignment and alignment extension.

3.1 Using Salmon

As mentioned above, there are two "modes" of operation for Salmon. The first, requires you to build an index for the transcriptome, but then subsequently processes reads directly. The second mode simply requires you to provide a FASTA file of the transcriptome and a `.sam` or `.bam` file containing a set of alignments.

Note: Read / alignment order

Salmon, like eXpress[1], uses a streaming inference method to perform transcript-level quantification. One of the fundamental assumptions of such inference methods is that observations (i.e. reads or alignments) are made "at random". This means, for example, that alignments should **not** be sorted by target or position. If your reads or alignments do not appear in a random order with respect to the target transcripts, please randomize / shuffle them before performing quantification with Salmon.

Note: Number of Threads

The number of threads that Salmon can effectively make use of depends upon the mode in which it is being run. In alignment-based mode, the main bottleneck is in parsing and decompressing the input BAM file. We make use of the Staden IO library for SAM/BAM/CRAM I/O (CRAM is, in theory, supported, but has not been thorougly tested). This means that multiple threads can be effectively used to aid in BAM decompression. However, we find that throwing more than a few threads at file decompression does not result in increased processing speed. Thus, alignment-based Salmon will only ever allocate up to 4 threads to file decompression, with the rest being allocated to quantification. If these threads are starved, they will sleep (the quantification threads do not busy wait), but there is a point beyond which allocating more threads will not speed up alignment-based quantification. We find that allocating 8 — 12 threads results in the maximum speed, threads allocated above this limit will likely spend most of their time idle / sleeping.

For quasi-mapping-based Salmon, the story is somewhat different. Generally, performance continues to improve as more threads are made available. This is because the determiniation of the potential mapping locations of each read is, generally, the slowest step in quasi-mapping-based quantification. Since this process is trivially parallelizable (and well-parallelized within Salmon), more threads generally equates to faster quantification. However, there may still be a limit to the return on invested threads, when Salmon can begin to process fragments more quickly than they can be provided via the parser.

3.2 Preparing transcriptome indices (mapping-based mode)

One of the novel and innovative features of Salmon is its ability to accurately quantify transcripts using *quasi-mapping*, with or without mapping validation. Quasi-mapping is typically **much** faster to compute than traditional (or full) alignments. More details about quasi-mappings, and how they are computed, can be found here.

If you want to use Salmon in mapping-based mode, then you first have to build an Salmon index for your transcriptome. Assume that `transcripts.fa` contains the set of transcripts you wish to quantify. First, you run the Salmon indexer:

```
> ./bin/salmon index -t transcripts.fa -i transcripts_index -k 31
```

This will build the quasi-mapping-based index, using an auxiliary k-mer hash over k-mers of length 31. While quasi-mapping will make used of arbitrarily long matches between the query and reference, the k size selected here will act as the *minimum* acceptable length for a valid match. Thus, a smaller value of k may slightly improve sensitivty. We find that a k of 31 seems to work well for reads of 75bp or longer, but you might consider a smaller k if you plan to deal

[1] Roberts, Adam, and Lior Pachter. "Streaming fragment assignment for real-time analysis of sequencing experiments." Nature Methods 10.1 (2013): 71-73.

with shorter reads. Also, a shoter value of *k* may improve sensitivity even more when using the *–validateMappings* flag. So, if you are seeing a smaller mapping rate than you might expect, consider building the index with a slightly smaller *k*.

3.3 Quantifying in mapping-based mode

Then, you can quantify any set of reads (say, paired-end reads in files *reads1.fq* and *reads2.fq*) directly against this index using the Salmon `quant` command as follows:

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -1 reads1.fq -2 reads2.fq --
↪validateMappings -o transcripts_quant
```

If you are using single-end reads, then you pass them to Salmon with the `-r` flag like:

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -r reads.fq --validateMappings␣
↪-o transcripts_quant
```

Note: Order of command-line parameters

The library type `-l` should be specified on the command line **before** the read files (i.e. the parameters to `-1` and `-2`, or `-r`). This is because the contents of the library type flag is used to determine how the reads should be interpreted.

You can, of course, pass a number of options to control things such as the number of threads used or the different cutoffs used for counting reads. Just as with the alignment-based mode, after Salmon has finished running, there will be a directory called `salmon_quant`, that contains a file called `quant.sf` containing the quantification results.

3.3.1 Providing multiple read files to Salmon

Often, a single library may be split into multiple FASTA/Q files. Also, sometimes one may wish to quantify multiple replicates or samples together, treating them as if they are one library. Salmon allows the user to provide a *space-separated* list of read files to all of it's options that expect input files (i.e. `-r`, `-1`, `-2`). When the input is paired-end reads, the order of the files in the left and right lists must be the same. There are a number of ways to provide salmon with multiple read files, and treat these as a single library. For the examples below, assume we have two replicates `lib_1` and `lib_2`. The left and right reads for `lib_1` are `lib_1_1.fq` and `lib_1_2.fq`, respectively. The left and right reads for `lib_2` are `lib_2_1.fq` and `lib_2_2.fq`, respectively. The following are both valid ways to input these reads to Salmon:

```
> salmon quant -i index -l IU -1 lib_1_1.fq lib_2_1.fq -2 lib_1_2.fq lib_2_2.fq --
↪validateMappings -o out

> salmon quant -i index -l IU -1 <(cat lib_1_1.fq lib_2_1.fq) -2 <(cat lib_1_2.fq lib_
↪2_2.fq) --validateMappings -o out
```

Similarly, both of these approaches can be adopted if the files are gzipped as well:

```
> salmon quant -i index -l IU -1 lib_1_1.fq.gz lib_2_1.fq.gz -2 lib_1_2.fq.gz lib_2_2.
↪fq.gz --validateMappings -o out

> salmon quant -i index -l IU -1 <(gunzip -c lib_1_1.fq.gz lib_2_1.fq.gz) -2 <(gunzip␣
↪-c lib_1_2.fq.gz lib_2_2.fq.gz) --validateMappings -o out
```

In each pair of commands, the first command lets Salmon natively parse the files, while the latter command creates, on-the-fly, an input stream that consists of the concatenation of both files. Both methods work, and are acceptable ways to merge the files. The latter method (i.e. process substitution) allows more complex processing to be done to the reads in the substituted process before they are passed to Salmon as input, and thus, in some situations, is more versatile.

Note: Interleaved FASTQ files

Salmon does not currently have built-in support for interleaved FASTQ files (i.e., paired-end files where both pairs are stored in the same file). We provide a script that can be used to run salmon with interleaved input. However, this script assumes that the input reads are perfectly synchronized. That is, the input cannot contain any un-paired reads.

3.4 Quantifying in alignment-based mode

Say that you've prepared your alignments using your favorite aligner and the results are in the file `aln.bam`, and assume that the sequence of the transcriptome you want to quantify is in the file `transcripts.fa`. You would run Salmon as follows:

```
> ./bin/salmon quant -t transcripts.fa -l <LIBTYPE> -a aln.bam -o salmon_quant
```

The `<LIBTYPE>` parameter is described below and is shared between both modes of Salmon. After Salmon has finished running, there will be a directory called `salmon_quant`, that contains a file called `quant.sf`. This contains the quantification results for the run, and the columns it contains are similar to those of Sailfish (and self-explanatory where they differ).

For the full set of options that can be passed to Salmon in its alignment-based mode, and a description of each, run `salmon quant --help-alignment`.

Note: Genomic vs. Transcriptomic alignments

Salmon expects that the alignment files provided are with respect to the transcripts given in the corresponding fasta file. That is, Salmon expects that the reads have been aligned directly to the transcriptome (like RSEM, eXpress, etc.) rather than to the genome (as does, e.g. Cufflinks). If you have reads that have already been aligned to the genome, there are currently 3 options for converting them for use with Salmon. First, you could convert the SAM/BAM file to a FAST{A/Q} file and then use the lightweight-alignment-based mode of Salmon described below. Second, given the converted FASTA{A/Q} file, you could re-align these converted reads directly to the transcripts with your favorite aligner and run Salmon in alignment-based mode as described above. Third, you could use a tool like sam-xlate to try and convert the genome-coordinate BAM files directly into transcript coordinates. This avoids the necessity of having to re-map the reads. However, we have very limited experience with this tool so far.

Multiple alignment files

If your alignments for the sample you want to quantify appear in multiple .bam/.sam files, then you can simply provide the Salmon `-a` parameter with a (space-separated) list of these files. Salmon will automatically read through these one after the other quantifying transcripts using the alignments contained therein. However, it is currently the case that these separate files must (1) all be of the same library type and (2) all be aligned with respect to the same reference (i.e. the @SQ records in the header sections must be identical).

3.5 Description of important options

Salmon exposes a number of useful optional command-line parameters to the user. The particularly important ones are explained here, but you can always run `salmon quant -h` to see them all.

3.5.1 -p / --threads

The number of threads that will be used for quasi-mapping, quantification, and bootstrapping / posterior sampling (if enabled). Salmon is designed to work well with many threads, so, if you have a sufficient number of processors, larger values here can speed up the run substantially.

Note: Default number of threads

The default behavior is for Salmon to probe the number of available hardware threads and to use this number. Thus, if you want to use fewer threads (e.g., if you are running multiple instances of Salmon simultaneously), you will likely want to set this option explicitly in accordance with the desired per-process resource usage.

3.5.2 --dumpEq

If Salmon is passed the `--dumpEq` option, it will write a file in the auxiliary directory, called `eq_classes.txt` that contains the equivalence classes and corresponding counts that were computed during quasi-mapping. The file has a format described in *Equivalence class file*.

3.5.3 --incompatPrior

This parameter governs the *a priori* probability that a fragment mapping or aligning to the reference in a manner incompatible with the prescribed library type is nonetheless the correct mapping. Note that Salmon sets this value, by default, to a small but *non-zero* probability. This means that if an incompatible mapping is the *only* mapping for a fragment, Salmon will still assign this fragment to the transcript. This default behavior is different than programs like RSEM, which assign incompatible fragments a 0 probability (i.e., incompatible mappings will be discarded). If you wish to obtain this behavior, so that only compatible mappings will be considered, you can set `--incompatPrior 0.0`. This will cause Salmon to only consider mappings (or alignments) that are compatible with the prescribed or inferred library type.

3.5.4 --fldMean

Note : This option is only important when running Salmon with single-end reads.

Since the empirical fragment length distribution cannot be estimated from the mappings of single-end reads, the `--fldMean` allows the user to set the expected mean fragment lenth of the sequencing library. This value will affect the effective length correction, and hence the estimated effective lengths of the transcripts and the TPMs. The value passed to `--fldMean` will be used as the mean of the assumed fragment length distribution (which is modeled as a truncated Gaussian with a standard deviation given by `--fldSD`).

3.5.5 --fldSD

Note : This option is only important when running Salmon with single-end reads.

Since the empirical fragment length distribution cannot be estimated from the mappings of single-end reads, the `--fldSD` allows the user to set the expected standard deviation of the fragment lenth distribution of the sequencing library. This value will affect the effective length correction, and hence the estimated effective lengths of the transcripts and the TPMs. The value passed to `--fldSD` will be used as the standard deviation of the assumed fragment length distribution (which is modeled as a truncated Gaussan with a mean given by `--fldMean`).

3.5.6 --validateMappings

One potential artifact that may arise from *alignment-free* mapping techniques is *spurious mappings*. These may either be reads that do not arise from some target being quantified, but nonetheless exhibit some match against them (e.g. contaminants) or, more commonly, mapping a read to a larger set of quantification targets than would be supported by an optimal or near-optimal alignment. Further, such mapping techniques can manifest a lack of sensitivity, where they fail to find the optimal (or all equally-optimal) mapping positions for a fragment.

If you pass the `--validateMappings` flag to Salmon, it will run an extension alignment dynamic program on the quasi-mappings it produces. The alignment procedure used to validate these mappings makes use of the highly-efficient and SIMD-parallelized ksw2[6] library. Moreover, Salmon makes use of an intelligent alignment cache to avoid re-computing alignment scores against redundant transcript sequences (e.g. when a read maps to the same exon in multiple different transcripts). The exact parameters used for scoring alignments, and the cutoff used for which mappings should be reported at all, are controllable by parameters described below.

This parameter should be used in conjunction with the range factorization option `--rangeFactorizationBins`, and can lead to improved quantification estimates. It is worth noting that this also makes quantification more sensitive to low-quality reads, so that e.g. quality trimming may become more important before processing reads using this option.

3.5.7 --minScoreFraction

This value controls the minimum allowed score for a mapping to be considered valid. It matters only when `--validateMappings` has been passed to Salmon. The maximum possible score for a fragment is `ms = read_len * ma` (or `ms = (left_read_len + right_read_len) * ma` for paired-end reads). The argument to `--minScoreFraction` determines what fraction of the maximum score `s` a mapping must achieve to be potentially retained. For a minimum score fraction of `f`, only mappings with a score > `f * s` will be kept. Mappings with lower scores will be considered as low-quality, and will be discarded.

It is worth noting that mapping validation uses extension alignment. This means that the read need not map end-to-end. Instead, the score of the mapping will be the position along the alignment with the highest score. This is the score which must reach the fraction threshold for the read to be considered as valid.

3.5.8 --ma

This value should be a positive (typically small) integer. It controls the score given to a match in the alignment between the query (read) and the reference.

3.5.9 --mp

This value should be a negative (typically small) integer. It controls the score given to a mismatch in the alignment between the query (read) and the reference.

3.5.10 --go

This value should be a positive (typically small) integer. It controls the score penalty attributed to an alignment for each new gap that is opened. The alignment score computed uses an affine gap penalty, so the penalty of a gap is `go + l * ge` where l is the gap length. The value of `go` should typically be larger than that of `ge`.

3.5.11 --ge

This value should be a positive (typically small) integer. It controls the score penalty attributed to the extension of a gap in an alignment. The alignment score computed uses an affine gap penalty, so the penalty of a gap is `go + l * ge` where l is the gap length. The value of `ge` should typically be smaller than that of `go`.

3.5.12 --rangeFactorizationBins

The range-factorization feature allows using a data-driven likelihood factorization, which can improve quantification accuracy on certain classes of "difficult" transcripts. Currently, this feature interacts best (i.e., yields the most considerable improvements) when either (1) using alignment-based mode and simultaneously enabling error modeling with `--useErrorModel` or (2) when enabling `--validateMappings` in quasi-mapping-based mode. The argument to this option is a positive integer `x`, that determines fidelity of the factorization. The larger `x`, the closer the factorization to the un-factorized likelihood, but the larger the resulting number of equivalence classes. A value of 1 corresponds to salmon's traditional rich equivalence classes. We recommend 4 as a reasonable parameter for this option (it is what was used in the range-factorization paper).

3.5.13 --useEM

Use the "standard" EM algorithm to optimize abundance estimates instead of the variational Bayesian EM algorithm. The details of the VBEM algorithm can be found in[3]. While both the standard EM and the VBEM produce accurate abundance estimates, there are some trade-offs between the approaches. Specifically, the sparsity of the VBEM algorithm depends on the prior that is chosen. When the prior is small, the VBEM tends to produce a sparser solution than the EM algorithm, while when the prior is relatively larger, it tends to estimate more non-zero abundances than the EM algorithm. It is an active research effort to analyze and understand all the tradeoffs between these different optimization approaches. Also, the VBEM tends to converge after fewer iterations, so it may result in a shorter runtime; especially if you are computing many bootstrap samples.

The default prior used in the VB optimization is a *per-nucleotide* prior of 1e-5 reads per-nucleotide. This means that a transcript of length 100000 will have a prior count of 1 fragment, while a transcript of length 50000 will have a prior count of 0.5 fragments, etc. This behavior can be modified in two ways. First, the prior itself can be modified via Salmon's `--vbPrior` option. The argument to this option is the value you wish to place as the *per-nucleotide* prior. Additonally, you can modify the behavior to use a *per-transcript* rather than a *per-nucleotide* prior by passing the flag `--perTranscriptPrior` to Salmon. In this case, whatever value is set by `--vbPrior` will be used as the transcript-level prior, so that the prior count is no longer dependent on the transcript length. However, the default behavior of a *per-nucleotide* prior is recommended when using VB optimization.

Note: Choosing between EM and VBEM algorithms

As mentioned above, a thorough comparison of all of the benefits and detriments of the different algorithms is an ongoing area of research. However, preliminary testing suggests that the sparsity-inducing effect of running the VBEM with a small prior may lead, in general, to more accurate estimates (the current testing was performed mostly through simulation). Hence, the VBEM is the default, and the standard EM algorithm is accessed via the *–useEM* flag.

---

[3] Patro, Rob, et al. "Salmon provides fast and bias-aware quantification of transcript expression." Nature Methods (2017). Advanced Online Publication. doi: 10.1038/nmeth.4197..

3.5.14 --numBootstraps

Salmon has the ability to optionally compute bootstrapped abundance estimates. This is done by resampling (with replacement) from the counts assigned to the fragment equivalence classes, and then re-running the optimization procedure, either the EM or VBEM, for each such sample. The values of these different bootstraps allows us to assess technical variance in the main abundance estimates we produce. Such estimates can be useful for downstream (e.g. differential expression) tools that can make use of such uncertainty estimates. This option takes a positive integer that dictates the number of bootstrap samples to compute. The more samples computed, the better the estimates of varaiance, but the more computation (and time) required.

3.5.15 --numGibbsSamples

Just as with the bootstrap procedure above, this option produces samples that allow us to estimate the variance in abundance estimates. However, in this case the samples are generated using posterior Gibbs sampling over the fragment equivalence classes rather than bootstrapping. We are currently analyzing these different approaches to assess the potential trade-offs in time / accuracy. The `--numBootstraps` and `--numGibbsSamples` options are mutually exclusive (i.e. in a given run, you must set at most one of these options to a positive integer.)

3.5.16 --seqBias

Passing the `--seqBias` flag to Salmon will enable it to learn and correct for sequence-specific biases in the input data. Specifically, this model will attempt to correct for random hexamer priming bias, which results in the preferential sequencing of fragments starting with certain nucleotide motifs. By default, Salmon learns the sequence-specific bias parameters using 1,000,000 reads from the beginning of the input. If you wish to change the number of samples from which the model is learned, you can use the `--numBiasSamples` parameter. Salmon uses a variable-length Markov Model (VLMM) to model the sequence specific biases at both the 5' and 3' end of sequenced fragments. This methodology generally follows that of Roberts et al.[2], though some details of the VLMM differ.

Note: This sequence-specific bias model is substantially different from the bias-correction methodology that was used in Salmon versions prior to 0.6.0. This model specifically accounts for sequence-specific bias, and should not be prone to the over-fitting problem that was sometimes observed using the previous bias-correction methodology.

3.5.17 --gcBias

Passing the `--gcBias` flag to Salmon will enable it to learn and correct for fragment-level GC biases in the input data. Specifically, this model will attempt to correct for biases in how likely a sequence is to be observed based on its internal GC content.

You can use the FASTQC software followed by MultiQC with transcriptome GC distributions to check if your samples exhibit strong GC bias, i.e. under-representation of some sub-sequences of the transcriptome. If they do, we obviously recommend using the `--gcBias` flag. Or you can simply run Salmon with `--gcBias` in any case, as it does not impair quantification for samples without GC bias, it just takes a few more minutes per sample. For samples with moderate to high GC bias, correction for this bias at the fragment level has been shown to reduce isoform quantification errors[4][3].

This bias is distinct from the primer biases learned with the `--seqBias` option. Though these biases are distinct, they are not completely independent. When both `--seqBias` and `--gcBias` are enabled, Salmon will learn a conditional fragment-GC bias model. By default, Salmon will learn 3 different fragment-GC bias models based on the GC content of the fragment start and end contexts, though this number of conditional models can be changed with the

[2] Roberts, Adam, et al. "Improving RNA-Seq expression estimates by correcting for fragment bias." Genome Biology 12.3 (2011): 1.

[4] Love, Michael I., Hogenesch, John B., Irizarry, Rafael A. "Modeling of RNA-seq fragment sequence bias reduces systematic errors in transcript abundance estimation." Nature Biotechnology 34.12 (2016). doi: 10.1038/nbt.368.2..

(*hidden*) option `--conditionalGCBins`. Likewise, the number of distinct fragment GC bins used to model the GC bias can be changed with the (*hidden*) option `--numGCBins`.

Note : In order to speed up the evaluation of the GC content of arbitrary fragments, Salmon pre-computes and stores the cumulative GC count for each transcript. This requires an extra 4-bytes per nucleotide. While this extra memory usage should normally be minor, it can nonetheless be controlled with the `--reduceGCMemory` option. This option replaces the per-nucleotide GC count with a rank-select capable bit vector, reducing the memory overhead from 4-bytes per nucleotide to ~1.25 bits, while being only marginally slower).

3.5.18 --posBias

Passing the `--posBias` flag to Salmon will enable modeling of a position-specific fragment start distribution. This is meant to model non-uniform coverage biases that are sometimes present in RNA-seq data (e.g. 5' or 3' positional bias). Currently, a small and fixed number of models are learned for different length classes of transcripts, as is done in Roberts et al.[2]. *Note*: The positional bias model is relatively new, and is still undergoing testing. It replaces the previous *–useFSPD* option, which is now deprecated. This feature should be considered as *experimental* in the current release.

3.5.19 --biasSpeedSamp

When evaluating the bias models (the GC-fragment model specifically), Salmon must consider the probability of generating a fragment of every possible length (with a non-trivial probability) from every position on every transcript. This results in a process that is quadratic in the length of the transcriptome — though each evaluation itself is efficient and the process is highly parallelized.

It is possible to speed this process up by a multiplicative factor by considering only every i^{th} fragment length, and interploating the intermediate results. The `--biasSpeedSamp` option allows the user to set this sampling factor. Larger values speed up effective length correction, but may decrease the fidelity of bias modeling. However, reasonably small values (e.g. 10 or less) should have only a minor effect on the computed effective lengths, and can considerably speed up effective length correction on large transcriptomes. The default value for `--biasSpeedSamp` is 5.

3.5.20 --writeUnmappedNames

Passing the `--writeUnmappedNames` flag to Salmon will tell Salmon to write out the names of reads (or mates in paired-end reads) that do not map to the transcriptome. When mapping paired-end reads, the entire fragment (both ends of the pair) are identified by the name of the first read (i.e. the read appearing in the `_1` file). Each line of the umapped reads file contains the name of the unmapped read followed by a simple flag that designates *how* the read failed to map completely. For single-end reads, the only valid flag is `u` (unmapped). However, for paired-end reads, there are a number of different possibilities, outlined below:

```
u   = The entire pair was unmapped. No mappings were found for either the left or␣
↪right read.
m1  = Left orphan (mappings were found for the left (i.e. first) read, but not the␣
↪right).
m2  = Right orphan (mappinds were found for the right read, but not the left).
m12 = Left and right orphans. Both the left and right read mapped, but never to the␣
↪same transcript.
```

By reading through the file of unmapped reads and selecting the appropriate sequences from the input FASTA/Q files, you can build an "unmapped" file that can then be used to investigate why these reads may not have mapped (e.g. poor quality, contamination, etc.). Currently, this process must be done independently, but future versions of Salmon may provide a script to generate this unmapped FASTA/Q file from the unmapped file and the original inputs.

3.5.21 --writeMappings

Passing the --writeMappings argument to Salmon will have an effect only in mapping-based mode and *only when using a quasi-index*. When executed with the --writeMappings argument, Salmon will write out the mapping information that it then processes to quantify transcript abundances. The mapping information will be written in a SAM compatible format. If no options are provided to this argument, then the output will be written to stdout (so that e.g. it can be piped to samtools and directly converted into BAM format). Otherwise, this argument can optionally be provided with a filename, and the mapping information will be written to that file. **Note:** Because of the way that the boost options parser that we use works, and the fact that --writeMappings has an implicit argument of stdout, if you provide an explicit argument to --writeMappings, you must do so with the syntax --writeMappings=<outfile> rather than the synatx --writeMappings <outfile>. This is a due to a limitation of the parser in how the latter could be interpreted.

Note: Compatible mappings

The mapping information is computed and written *before* library type compatibility checks take place, thus the mapping file will contain information about all mappings of the reads considered by Salmon, even those that may later be filtered out due to incompatibility with the library type.

3.6 What's this LIBTYPE?

Salmon, has the user provide a description of the type of sequencing library from which the reads come, and this contains information about e.g. the relative orientation of paired end reads. As of version 0.7.0, Salmon also has the ability to automatically infer (i.e. guess) the library type based on how the first few thousand reads map to the transcriptome. To allow Salmon to automatically infer the library type, simply provide -l A or --libType A to Salmon. Even if you allow Salmon to infer the library type for you, you should still read the section below, so that you can interpret how Salmon reports the library type it discovers.

Note: Automatic library type detection in alignment-based mode

The implementation of this feature involves opening the BAM file, peaking at the first record, and then closing it to determine if the library should be treated as single-end or paired-end. Thus, *in alignment-based mode* automatic library type detection will not work with an input stream. If your input is a regular file, everything should work as expected; otherwise, you should provide the library type explicitly in alignment-based mode.

Also the automatic library type detection is performed *on the basis of the alignments in the file*. Thus, for example, if the upstream aligner has been told to perform strand-aware mapping (i.e. to ignore potential alignments that don't map in the expected manner), but the actual library is unstranded, automatic library type detection cannot detect this. It will attempt to detect the library type that is most consistent *with the alignment that are provided*.

The library type string consists of three parts: the relative orientation of the reads, the strandedness of the library, and the directionality of the reads.

The first part of the library string (relative orientation) is only provided if the library is paired-end. The possible options are:

```
I = inward
O = outward
M = matching
```

The second part of the read library string specifies whether the protocol is stranded or unstranded; the options are:

```
S = stranded
U = unstranded
```

If the protocol is unstranded, then we're done. The final part of the library string specifies the strand from which the read originates in a strand-specific protocol — it is only provided if the library is stranded (i.e. if the library format string is of the form S). The possible values are:

```
F = read 1 (or single-end read) comes from the forward strand
R = read 1 (or single-end read) comes from the reverse strand
```

An example of some library format strings and their interpretations are:

```
IU (an unstranded paired-end library where the reads face each other)
```

```
SF (a stranded single-end protocol where the reads come from the forward strand)
```

```
OSR (a stranded paired-end protocol where the reads face away from each other,
     read1 comes from reverse strand and read2 comes from the forward strand)
```

Note: Strand Matching

Above, when it is said that the read "comes from" a strand, we mean that the read should align with / map to that strand. For example, for libraries having the `OSR` protocol as described above, we expect that read1 maps to the reverse strand, and read2 maps to the forward strand.

For more details on the library type, see *Fragment Library Types*.

3.7 Output

For details of Salmon's different output files and their formats see *Salmon Output File Formats*.

3.8 Misc

Salmon, in *quasi-mapping*-based mode, can accept reads from FASTA/Q format files, or directly from gzipped FASTA/Q files (the ability to accept compressed files directly is a feature of Salmon 0.7.0 and higher). If your reads are compressed in a different format, you can still stream them directly to Salmon by using process substitution. Say in the *quasi-mapping*-based Salmon example above, the reads were actually in the files `reads1.fa.bz2` and `reads2.fa.bz2`, then you'd run the following command to decompress the reads "on-the-fly":

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -1 <(bunzip2 -c reads1.fa.gz) -
↪2 <(bunzip2 -c reads2.fa.bz2) -o transcripts_quant
```

and the bzipped files will be decompressed via separate processes and the raw reads will be fed into Salmon. Actually, you can use this same process even with gzip compressed reads (replacing `bunzip2` with `gunzip` or `pigz -d`). Depending on the number of threads and the exact configuration, this may actually improve Salmon's running time, since the reads are decompressed concurrently in a separate process when you use process substitution.

Finally, the purpose of making this software available is for people to use it and provide feedback. The paper describing this method is published in Nature Methods. If you have something useful to report or just some interesting ideas or suggestions, please contact us (*rob.patro@cs.stonybrook.edu* and/or *carlk@cs.cmu.edu*). If you encounter any bugs, please file a *detailed* bug report at the Salmon GitHub repository.

3.9 References

CHAPTER 4

Alevin

Alevin is a tool — integrated with the salmon software — that introduces a family of algorithms for quantification and analysis of 3' tagged-end single-cell sequencing data. Currently alevin supports the following two major droplet based single-cell protocols:

1. Drop-seq
2. 10x-Chromium v1/2/3

Alevin works under the same indexing scheme (as salmon) for the reference, and consumes the set of FASTA/Q files(s) containing the Cellular Barcode(CB) + Unique Molecule identifier (UMI) in one read file and the read sequence in the other. Given just the transcriptome and the raw read files, alevin generates a cell-by-gene count matrix (in a fraction of the time compared to other tools).

Alevin works in two phases. In the first phase it quickly parses the read file containing the CB and UMI information to generate the frequency distribution of all the observed CBs, and creates a lightweight data-structure for fast-look up and correction of the CB. In the second round, alevin utilizes the read-sequences contained in the files to map the reads to the transcriptome, identify potential PCR/sequencing errors in the UMIs, and performs hybrid de-duplication while accounting for UMI collisions. Finally, a post-abundance estimation CB whitelisting procedure is done and a cell-by-gene count matrix is generated.

4.1 Using Alevin

Alevin requires the following minimal set of necessary input parameters (generally providing the flags *in that order* is recommended):

- `-l`: library type (same as salmon), we recommend using *ISR* for both Drop-seq and 10x-v2 chemistry.
- `-1`: CB+UMI file(s), alevin requires the path to the *FASTQ* file containing CB+UMI raw sequences to be given under this command line flag. Alevin also supports parsing of data from multiple files as long as the order is the same as in *-2* flag.
- `-2`: Read-sequence file(s), alevin requires the path to the *FASTQ* file containing raw read-sequences to be given under this command line flag. Alevin also supports parsing of data from multiple files as long as the order is the same as in *-1* flag.

- `--dropseq / --chromium`: the protocol, this flag tells the type of single-cell protocol of the input sequencing-library.
- `-i`: index, file containing the salmon index of the reference transcriptome, as generated by *salmon index* command.
- `-p`: number of threads, the number of threads which can be used by alevin to perform the quantification, by default alevin utilizes *all* the available threads in the system, although we recommend using ~10 threads which in our testing gave the best memory-time trade-off.
- `-o`: output, path to folder where the output gene-count matrix (along with other meta-data) would be dumped.
- `--tgMap`: transcript to gene map file, a tsv (tab-separated) file — with *no header*, containing two columns mapping of each transcript present in the reference to the corresponding gene (the first column is a transcript and the second is the corresponding gene).

Once all the above requirement are satisfied, alevin can be run using the following command:

```
> salmon alevin -l ISR -1 cb.fastq.gz -2 reads.fastq.gz --chromium  -i salmon_index_
↪directory -p 10 -o alevin_output --tgMap txp2gene.tsv
```

4.2 Providing multiple read files to Alevin

Often, a single library may be split into multiple FASTA/Q files. Also, sometimes one may wish to quantify multiple replicates or samples together, treating them as if they are one library. Alevin allows the user to provide a *space-separated* list of files to all of it's options that expect input files (i.e. `-1`, `-2`). The order of the files in the left and right lists must be the same. There are a number of ways to provide alevin with multiple CB and read files, and treat these as a single library. For the examples below, assume we have two replicates `lib_A` and `lib_B`. The left and right reads for `lib_A` are `lib_A_cb.fq` and `lib_A_reads.fq`, respectively. The left and right reads for `lib_B` are `lib_B_cb.fq` and `lib_B_read.fq`, respectively. The following are both valid ways to input these reads to alevin:

```
> salmon alevin -lISR -1 lib_A_cb.fq lib_B_cb.fq -2 lib_A_read.fq lib_B_read.fq
```

Similarly, both of these approaches can be adopted if the files are gzipped as well:

```
> salmon alevin -l ISR -1 lib_A_cb.fq.gz lib_B_cb.fq.gz -2 lib_A_read.fq.gz lib_B_
↪read.fq.gz
```

Note: Don't provide data through input stream

To keep the time-memory trade-off within acceptable bounds, alevin performs multiple passes over the Cellular Barcode file. Alevin goes through the barcode file once by itself, and then goes through both the barcode and read files in unison to assign reads to cells using the initial barcode mapping. Since the pipe or the input stream can't be reset to read from the beginning again, alevin can't read in the barcodes, and might crash.

4.3 Description of important options

Alevin exposes a number of useful optional command-line parameters to the user. The particularly important ones are explained here, but you can always run `salmon alevin -h` to see them all.

4.3.1 -p / --numThreads

The number of threads that will be used for quantification. Alevin is designed to work well with many threads, so, if you have a sufficient number of processors, larger values here can speed up the run substantially. In our testing we found that usually 10 threads gives the best time-memory trade-off.

Note: Default number of threads

> The default behavior is for Alevin to probe the number of available hardware threads and to use this number.

Thus, if you want to use fewer threads (e.g., if you are running multiple instances of Salmon simultaneously), you will likely want to set this option explicitly in accordance with the desired per-process resource usage.

4.3.2 --whitelist

This is an optional argument, where user can explicitly specify the whitelist CB to use for cell detection and CB sequence correction. If not given, alevin generates its own set of putative CBs.

Note: Not 10x 724k whitelist

This flag does not use the biologically known whitelist provided by 10x, instead it's per experiment level whitelist file e.g. the file generated by cellranger with the name *barcodes.tsv*.

4.3.3 --noQuant

Generally used in parallel with `--dumpfq`. If Alevin is passed the `--noQuant` option, the pipeline will stop before starting the mapping. The general use-case is when we only need to concatenate the CB on the read-id of the second file and break the execution afterwards.

4.3.4 --noDedup

If Alevin is passed the `--noDedup` option, the pipeline only performs CB correction, maps the read-sequences to the transcriptome generating the interim data-structure of CB-EqClass-UMI-count. Used in parallel with `--dumpBarcodeEq` or `--dumpBfh` for the purposes of obtaining raw information or debugging.

4.3.5 --mrna

The list of mitochondrial genes which are to be used as a feature for CB whitelising naive Bayes classification.

4.3.6 --rrna

The list of ribosomal genes which are to be used as a feature for CB whitelising naive Bayes classification.

4.3.7 --useCorrelation

If activated, in CB whitelist classification alevin computes the cell-by-cell pearson correlation of each candidate CB with putative true set of CB. This flag can slow down alevin's processing.

4.3.8 --dumpfq

Generally used along with `--noQuant`. If activated, alevin will sequence correct the CB and attach the corrected CB sequence to the read-id in the second file and dumps the result to standard-out (`stdout`).

4.3.9 --dumpBfh

Alevin internally uses a potentially big data-structure to concisely maintain all the required information for quantification. This flags dumps the full CB-EqClass-UMI-count data-structure for the purposed of allowing raw data analysis and debugging.

4.3.10 --dumpFeatures

If activated, alevin dumps all the features used by the CB classification and their counts at each cell level. Generally, this is used for the purposes of debugging.

4.3.11 --dumpCsvCounts

This flags is used to internally convert the default binary format of alevin for gene-count matrix into a human readable csv (comma separated) format. The expression of all the gene in one cell is written in one row, while columns represents the genes.

4.3.12 --forceCells

Alevin performs a heuristic based initial CB white-listing by finding the knee in the distribution of the CB frequency. Although knee finding algorithm works pretty well in most of the case, it sometimes over shoot and results in very less number of CB. With this flag, by looking at the CB frequency distribution, a user can explicitly specify the number of CB to consider for initial white-listing.

4.3.13 --expectCells

Just like *forceCells* flag, it's yet another way of skipping the knee calculation heuristics, if it's failing. This command line flag uses the cellranger type white-listing procedure. As specified in their algorithm overview page, "All barcodes whose total UMI counts exceed m/10 are called as cells", where m is the frequency of the top 1% cells as specified by the parameter of this command line flag.

4.3.14 --numCellBootstraps

Alevin provides an estimate of the inferential uncertainty in the estimation of per cell level gene count matrix by performing bootstrapping of the reads in per-cell level equivalence classes. This command line flag informs Alevin to perform certain number of bootstrap and generate the mean and variance of the count matrix. This option generates three additional file, namely, *quants_mean_mat.gz*, *quants_var_mat.gz* and *quants_boot_rows.txt*. The format of the files stay the same as *quants_mat.gz* while the row order is saved in *quants_boot_rows.txt* and the column order is stays the same as in file *quants_mat_cols.txt*.

4.3.15 --debug

Alevin peforms intelligent white-listing downstream of the quantification pipeline and has to make some assumptions like chosing a fraction of reads to learn low confidence CB and in turn might erroneously exit – if the data results in no mapped or deduplicated reads to a CB in low confidence region. The problem doesn't happen when provided with external whitelist but if there is an error and the user is aware of this being just a warning, the error can be skipped by running Alevin with this flag.

4.3.16 --minScoreFraction

This value controls the minimum allowed score for a mapping to be considered valid. It matters only when `--validateMappings` has been passed to Salmon. The maximum possible score for a fragment is `ms = read_len * ma` (or `ms = (left_read_len + right_read_len) * ma` for paired-end reads). The argument to `--minScoreFraction` determines what fraction of the maximum score `s` a mapping must achieve to be potentially retained. For a minimum score fraction of `f`, only mappings with a score > `f * s` will be kept. Mappings with lower scores will be considered as low-quality, and will be discarded.

It is worth noting that mapping validation uses extension alignment. This means that the read need not map end-to-end. Instead, the score of the mapping will be the position along the alignment with the highest score. This is the score which must reach the fraction threshold for the read to be considered as valid.

4.4 Output

Typical 10x experiment can range form hundreds to tens of thousand of cells – resulting in huge size of the count-matrices. Traditionally single-cell tools dumps the Cell-v-Gene count matrix in various formats. Although, this itself is an open area of research but by default alevin dumps a per-cell level gene-count matrix in a binary-compressed format with the row and column indexes in a separate file.

A typical run of alevin will generate 4 files:

- *quants_mat.gz* – Compressed count matrix.
- *quants_mat_cols.txt* – Column Header (Gene-ids) of the matrix.
- *quants_mat_rows.txt* – Row Index (CB-ids) of the matrix.
- *quants_tier_mat.gz* – Tier categorization of the matrix.

Along with the Cell-v-Gene count matrix, alevin dumps a 3-fold categorization of each estimated count value of a gene(each cell disjointly) in the form of tiers. Tier 1 is the set of genes where all the reads are uniquely mapping. Tier 2 is genes that have ambiguously mapping reads, but connected to unique read evidence as well, that can be used by the EM to resolve the multimapping reads. Tier 3 is the genes that have no unique evidence and the read counts are, therefore, distributed between these genes according to an uninformative prior.

Alevin can also dump the count-matrix in a human readable – comma-separated-value (_CSV_) format, if given flag *–dumpCsvCounts* which generates a new output file called *quants_mat.csv*.

4.5 Misc

Finally, the purpose of making this software available is because we believe it may be useful for people dealing with single-cell RNA-seq data. We want the software to be as useful, robust, and accurate as possible. So, if you have any feedback — something useful to report, or just some interesting ideas or suggestions — please contact us (*asrivastava@cs.stonybrook.edu* and/or *rob.patro@cs.stonybrook.edu*). If you encounter any bugs, please file a *detailed* bug report at the Salmon GitHub repository.

4.6 References

CHAPTER 5

Salmon Output File Formats

5.1 Quantification File

Salmon's main output is its quantification file. This file is a plain-text, tab-separated file with a single header line (which names all of the columns). This file is named `quant.sf` and appears at the top-level of Salmon's output directory. The columns appear in the following order:

Name	Length	EffectiveLength	TPM	NumReads

Each subsequent row describes a single quantification record. The columns have the following interpretation.

- **Name** — This is the name of the target transcript provided in the input transcript database (FASTA file).
- **Length** — This is the length of the target transcript in nucleotides.
- **EffectiveLength** — This is the computed *effective* length of the target transcript. It takes into account all factors being modeled that will effect the probability of sampling fragments from this transcript, including the fragment length distribution and sequence-specific and gc-fragment bias (if they are being modeled).
- **TPM** — This is salmon's estimate of the relative abundance of this transcript in units of Transcripts Per Million (TPM). TPM is the recommended relative abundance measure to use for downstream analysis.
- **NumReads** — This is salmon's estimate of the number of reads mapping to each transcript that was quantified. It is an "estimate" insofar as it is the expected number of reads that have originated from each transcript given the structure of the uniquely mapping and multi-mapping reads and the relative abundance estimates for each transcript.

5.2 Command Information File

In the top-level quantification directory, there will be a file called `cmd_info.json`. This is a JSON format file that records the main command line parameters with which Salmon was invoked for the run that produced the output in this directory.

5.3 Auxiliary Files

The top-level quantification directory will contain an auxiliary directory called `aux_info` (unless the auxiliary directory name was overridden via the command line). This directory will have a number of files (and subfolders) depending on how salmon was invoked.

5.3.1 Meta information

The auxiliary directory will contain a JSON format file called `meta_info.json` which contains meta information about the run, including stats such as the number of observed and mapped fragments, details of the bias modeling etc. If Salmon was run with automatic inference of the library type (i.e. `--libType A`), then one particularly important piece of information contained in this file is the inferred library type. Most of the information recorded in this file should be self-descriptive.

5.3.2 Unique and ambiguous count file

The auxiliary directory also contains 2-column tab-separated file called `ambig_info.tsv`. This file contains information about the number of uniquely-mapping reads as well as the total number of ambiguously-mapping reads for each transcript. This file is provided mostly for exploratory analysis of the results; it gives some idea of the fraction of each transcript's estimated abundance that derives from ambiguously-mappable reads.

5.3.3 Observed library format counts

When run in *mapping-based* mode, the quantification directory will contain a file called `lib_format_counts.json`. This JSON file reports the number of fragments that had at least one mapping compatible with the designated library format, as well as the number that didn't. It also records the strand-bias that provides some information about how strand-specific the computed mappings were.

Finally, this file contains a count of the number of *mappings* that were computed that matched each possible library type. These are counts of *mappings*, and so a single fragment that maps to the transcriptome in more than one way may contribute to multiple library type counts. **Note**: This file is currently not generated when Salmon is run in alignment-based mode.

5.3.4 Fragment length distribution

The auxiliary directory will contain a file called `fld.gz`. This file contains an approximation of the observed fragment length distribution. It is a gzipped, binary file containing integer counts. The number of (signed, 32-bit) integers (with machine-native endianness) is equal to the number of bins in the fragment length distribution (1,001 by default — for fragments ranging in length from 0 to 1,000 nucleotides).

5.3.5 Sequence-specific bias files

If sequence-specific bias modeling was enabled, there will be 4 files in the auxiliary directory named `obs5_seq.gz`, `obs3_seq.gz`, `exp5_seq.gz`, `exp5_seq.gz`. These encode the parameters of the VLMM that were learned for the 5' and 3' fragment ends. Each file is a gzipped, binary file with the same format.

It begins with 3 32-bit signed integers which record the length of the context (window around the read start / end) that is modeled, follwed by the length of the context that is to the left of the read and the length of the context that is to the right of the read.

Next, the file contains 3 arrays of 32-bit signed integers (each of which have a length of equal to the context length recorded above). The first records the order of the VLMM used at each position, the second records the *shifts* and the *widths* required to extract each sub-context — these are implementation details.

Next, the file contains a matrix that encodes all VLMM probabilities. This starts with two signed integers of type `std::ptrdiff_t`. This is a platform-specific type, but on most 64-bit systems should correspond to a 64-bit signed integer. These numbers denote the number of rows (*nrow*) and columns (*ncol*) in the array to follow.

Next, the file contains an array of (*nrow* * *ncol*) doubles which represent a dense matrix encoding the probabilities of the VLMM. Each row corresponds to a possible preceeding sub-context, and each column corresponds to a position in the sequence context. Unused values (values where the length of the sub-context exceed the order of the model at that position) contain a 0. This array can be re-shaped into a matrix of the appropriate size.

Finally, the file contains the marginalized 0:sup:*th*-order probabilities (i.e. the probability of each nucleotide at each position in the context). This is stored as a 4-by-context length matrix. As before, this entry begins with two signed integers that give the number of rows and columns, followed by an array of doubles giving the marginal probabilities. The rows are in lexicographic order.

5.3.6 Fragment-GC bias files

If Salmon was run with fragment-GC bias correction enabled, the auxiliary directory will contain two files named `expected_gc.gz` and `observed_gc.gz`. These are gzipped binary files containing, respectively, the expected and observed fragment-GC content curves. These files both have the same form. They consist of a 32-bit signed int, *dtype* which specifies if the values to follow are in logarithmic space or not. Then, the file contains two signed integers of type `std::ptrdiff` which give the number of rows and columns of the matrix to follow. Finally, there is an array of *nrow* by *ncol* doubles. Each row corresponds to a conditional fragment GC distribution, and the number of columns is the number of bins in the learned (or expected) fragment-GC distribution.

5.3.7 Equivalence class file

If Salmon was run with the `--dumpEq` option, then a file called `eq_classes.txt` will exist in the auxiliary directory. The format of that file is as follows:

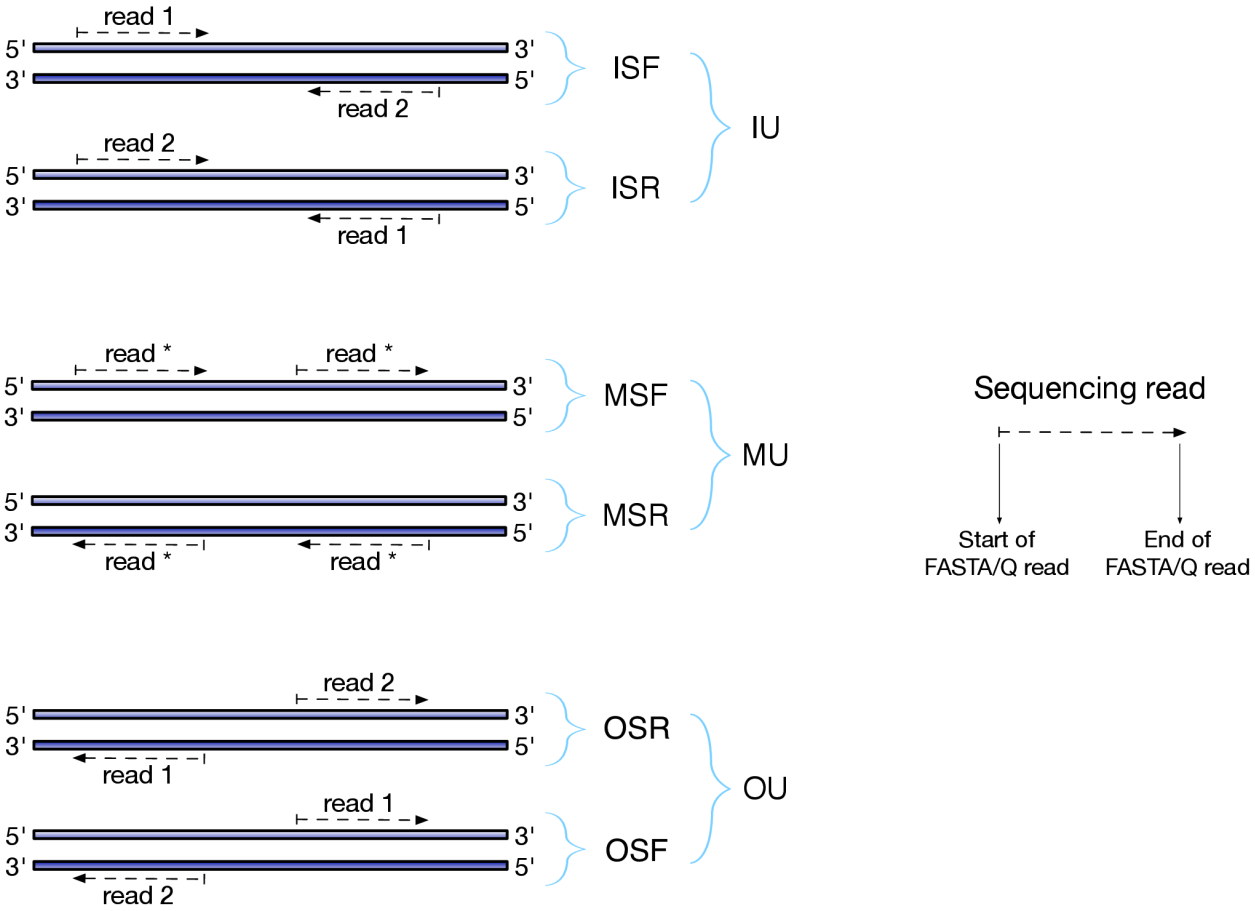
```
N (num transcripts)
M (num equiv classes)
tn_1
tn_2
...
tn_N
eq_1_size t_11 t_12 ... count
eq_2_size t_21 t_22 ... count
```

That is, the file begins with a line that contains the number of transcripts (say N) then a line that contains the number of equivalence classes (say M). It is then followed by N lines that list the transcript names — the order here is important, because the labels of the equivalence classes are given in terms of the ID's of the transcripts. The rank of a transcript in this list is the ID with which it will be labeled when it appears in the label of an equivalence class. Finally, the file contains M lines, each of which describes an equivalence class of fragments. The first entry in this line is the number of transcripts in the label of this equivalence class (the number of different transcripts to which fragments in this class map — call this k). The line then contains the k transcript IDs. Finally, the line contains the count of fragments in this equivalence class (how many fragments mapped to these transcripts). The values in each such line are tab separated.

CHAPTER 6

Fragment Library Types

There are numerous library preparation protocols for RNA-seq that result in sequencing reads with different characteristics. For example, reads can be single end (only one side of a fragment is recorded as a read) or paired-end (reads are generated from both ends of a fragment). Further, the sequencing reads themselves may be unstranded or strand-specific. Finally, paired-end protocols will have a specified relative orientation. To characterize the various different typs of sequencing libraries, we've created a miniature "language" that allows for the succinct description of the many different types of possible fragment libraries. For paired-end reads, the possible orientations, along with a graphical description of what they mean, are illustrated below:



The library type string consists of three parts: the relative orientation of the reads, the strandedness of the library, and the directionality of the reads.

The first part of the library string (relative orientation) is only provided if the library is paired-end. The possible options are:

```
I = inward
O = outward
M = matching
```

The second part of the read library string specifies whether the protocol is stranded or unstranded; the options are:

```
S = stranded
U = unstranded
```

If the protocol is unstranded, then we're done. The final part of the library string specifies the strand from which the read originates in a strand-specific protocol — it is only provided if the library is stranded (i.e. if the library format string is of the form S). The possible values are:

```
F = read 1 (or single-end read) comes from the forward strand
R = read 1 (or single-end read) comes from the reverse strand
```

So, for example, if you wanted to specify a fragment library of strand-specific paired-end reads, oriented toward each other, where read 1 comes from the forward strand and read 2 comes from the reverse strand, you would specify `-l ISF` on the command line. This designates that the library being processed has the type "ISF" meaning, **I**nward (the relative orientation), **S**tranded (the protocol is strand-specific), **F**orward (read 1 comes from the forward strand).

The single end library strings are a bit simpler than their pair-end counter parts, since there is no relative orientation of which to speak. Thus, the only possible library format types for single-end reads are `U` (for unstranded), `SF` (for strand-specific reads coming from the forward strand) and `SR` (for strand-specific reads coming from the reverse strand).

A few more examples of some library format strings and their interpretations are:

```
IU (an unstranded paired-end library where the reads face each other)
```

```
SF (a stranded single-end protocol where the reads come from the forward strand)
```

```
OSR (a stranded paired-end protocol where the reads face away from each other,
     read1 comes from reverse strand and read2 comes from the forward strand)
```

Note: Correspondence to TopHat library types

The popular TopHat RNA-seq read aligner has a different convention for specifying the format of the library. Below is a table that provides the corresponding sailfish/salmon library format string for each of the potential TopHat library types:

TopHat	Salmon (and Sailfish)	
	Paired-end	Single-end
`-fr-unstranded`	`-l IU`	`-l U`
`-fr-firststrand`	`-l ISR`	`-l SR`
`-fr-secondstrand`	`-l ISF`	`-l SF`

The remaining salmon library format strings are not directly expressible in terms of the TopHat library types, and so there is no direct mapping for them.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search