Sailfish Documentation

Release 0.6.4

Rob Patro, Carl Kingsford and Steve Mount

March 18, 2015

Contents

1 Requirements **3**

2 Installation **5**

3 Sailfish **7**
- 3.1 Indexing . . . 7
- 3.2 Quantification . . . 7
- 3.3 References . . . 8

4 Salmon **9**
- 4.1 Using Salmon . . . 9
- 4.2 Alignment-based mode . . . 10
- 4.3 Read-based mode . . . 10
- 4.4 What's this `LIBTYPE`? . . . 11
- 4.5 Output . . . 12
- 4.6 Misc . . . 12

5 Fragment Library Types **15**

6 Indices and tables **17**

Contents:

CHAPTER 1

Requirements

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

CHAPTER 2

Installation

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

```
> cd Sailfish
```

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

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

Sailfish 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 Sailfish 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").

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
```

To ensure that Sailfish has access to the appropriate libraries you should ensure that the PATH variabile contains <install_dir>/bin, and that LD_LIBRARY_PATH (or DYLD_FALLBACK_LIBRARY_PATH on OSX) contains <install_dir>/lib.

After the paths are set, 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

Sailfish

Sailfish is a tool for 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 sailfish is a fasta file containing your reference transcripts and a (set of) fasta/fastq file(s) containing your reads. Sailfish runs in two phases; indexing and quantification. The indexing step is independent of the reads, and only needs to be run once for a particular set of reference transcripts and choice of k (the k-mer size). 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 sailfish, see the manual.

3.1 Indexing

To generate the sailfish index for your reference set of transcripts, you should run the following command:

```
> sailfish index -t <ref_transcripts> -o <out_dir> -k <kmer_len>
```

This will build a sailfish index for k-mers of length `<kmer_len>` for the reference transcripts provided in the file `<ref_transcripts>` and place the index under the directory `<out_dir>`. There are additional options that can be passed to the sailfish indexer (e.g. the number of threads to use). These can be seen by executing the command `sailfish index -h`.

3.2 Quantification

Now that you have generated the sailfish index (say that it's the directory `<index_dir>` — this corresponds to the <out_dir> argument provided in the previous step), you can quantify the transcript expression for a given set of reads. To perform the quantification, you run a command like the following:

```
> sailfish quant -i <index_dir> -l "<libtype>" {-r <unmated> | -1 <mates1> -2 <mates2>} -o <quant_di
```

Where `<index_dir>` is, as described above, the location of the sailfish index, `<libtype>` is a string describing the format of the fragment (read) library (see *Fragment Library Types*), `<unmated>` is a list of files containing unmated reads, `<mates{1,2}>` are lists of files containg, respectively, the first and second mates of paired-end reads. Finally, `<quant_dir>` is the directory where the output should be written. Just like the indexing step, additional options are available, and can be viewed by running `sailfish quant -h`.

When the quantification step is finished, the directory `<quant_dir>` will contain a file named "quant.sf" (and, if bias correction is enabled, an additional file names "quant_bias_corrected.sf"). This file contains the result of the Sailfish quantification step. This file contains a number of columns (which are listed in the last of the header lines beginning with '#'). Specifically, the columns are (1) Transcript ID, (2) Transcript Length, (3) Transcripts per Million (TPM), (4) Reads Per Kilobase per Million mapped reads (RPKM), (5) K-mers Per Kilobase per Million mapped k-mers (KPKM), (6) Estimated number of k-mers (an estimate of the number of k-mers drawn from this transcript

given the transcript's relative abundance and length) and (7) Estimated number of reads (an estimate of the number of reads drawn from this transcript given the transcript's relative abnundance and length). The first two columns are self-explanatory, the next four are measures of transcript abundance and the final is a commonly used input for differential expression tools. The Transcripts per Million quantification number is computed as described in [1], and is meant as an estimate of the number of transcripts, per million observed transcripts, originating from each isoform. Its benefit over the K/RPKM measure is that it is independent of the mean expressed transcript length (i.e. if the mean expressed transcript length varies between samples, for example, this alone can affect differential analysis based on the K/RPKM.) The RPKM is a classic measure of relative transcript abundance, and is an estimate of the number of reads per kilobase of transcript (per million mapped reads) originating from each transcript. The KPKM should closely track the RPKM, but is defined for very short features which are larger than the chosen k-mer length but may be shorter than the read length. Typically, you should prefer the KPKM measure to the RPKM measure, since the k-mer is the most natural unit of coverage for Sailfish.

3.3 References

[1] Li, Bo, et al. "RNA-Seq gene expression estimation with read mapping uncertainty." Bioinformatics 26.4 (2010): 493-500.

CHAPTER 4

Salmon

Salmon is a tool for 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. Optinonally, 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 read-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.

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.

4.1 Using Salmon

As mentioned above, there are two "modes" of operation for Salmon. The first, like Sailfish, 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, 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 read-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 read-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. Specifically, writing to the mapping cache (see Misc below) is done via a single thread. With a huge number of quantification threads or in environments with a very slow disk, this may become the limiting step. If you're certain that you have more than the required number of observations, or if you have reason to suspect that your disk is particularly slow on writes, then you can disable the mapping cache (`--disableMappingCache`), and potentially increase the parallelizability of read-based salmon.

4.2 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 read-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).

4.3 Read-based mode

If you want to use salmon like sailfish, then you first have to build an salmon index for your transcriptome. Again, 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
```

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

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

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.

4.4 What's this `LIBTYPE`?

Salmon, like sailfish, 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. However, we've replaced the somewhat esoteric description of the library type with a simple set of strings; each of which represents a different type of read library. This new method of specifying the type of read library is being back-ported into Sailfish and will be available in the next release.

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*.

4.5 Output

Salmon writes its output in a simple tab-delimited file format. Any line that begins with a # is a comment line, and can be safely ignored. Salmon records the files and options passed to it in comments at the top of its output file. The last comment line gives the names of each of the data columns. The columns appear in the following order:

Name	Length	TPM	FPKM	NumReads

Each subsequent row described 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.
- **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.
- **FPKM** — This is salmon's estimate of the relative abundance of this transcript in units of Fragments Per Kilobase per Million mapped reads (FPKM). This relative abundance measure is proportional, within-sample, to the TPM measure. However, the TPM should generally be preferred to FPKM. This column is provided mostly for compatibility with tools that expect FPKM as input.
- **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. You can round these values to the nearest integer and use them directly as input to count-based methods like Deseq2 and EdgeR, among others.

4.6 Misc

Salmon deals with reading from compressed read files in the same way as sailfish — by using process substitution. Say in the read-based salmon example above, the reads were actually in the files `reads1.fa.gz` and `reads2.fa.gz`, then you'd run the following command to decompress the reads "on-the-fly":

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -1 <(gzcat reads1.fa.gz) -2 <(gzcat reads2.fa
```

and the gzipped files will be decompressed via separate processes and the raw reads will be fed into salmon.

Note: The Mapping Cache

Salmon requires a specific number of observations (fragments) to be observed before it will report its quantification results. If it doesn't see enough fragments when reading through the read files the first time, it will process the information again (don't worry; it's not double counting. The results from the first pass essentially become a "prior" for assigning the proper read counts in subsequent passes).

The first time the file is processed, the set of potential mappings for each fragment is written to a temporary file in an efficient binary format — this file is called the mapping cache. As soon as the required number of obvservations have been seen, salmon stops writing to the mapping cache (ensuring that the file size will not grow too large). However, for experiments with fewer than the required number of observations, the mapping cache is a significant optimization over reading through the raw set of reads multiple times. First, the work of determining the potential mapping locations for a read is only performed once, during the inital pass through the file. Second, since the mapping cache is implemented as a regular file on disk, the information contained within a file can be processed multiple times, even if the file itself is being produced via e.g. process substitution as in the example above.

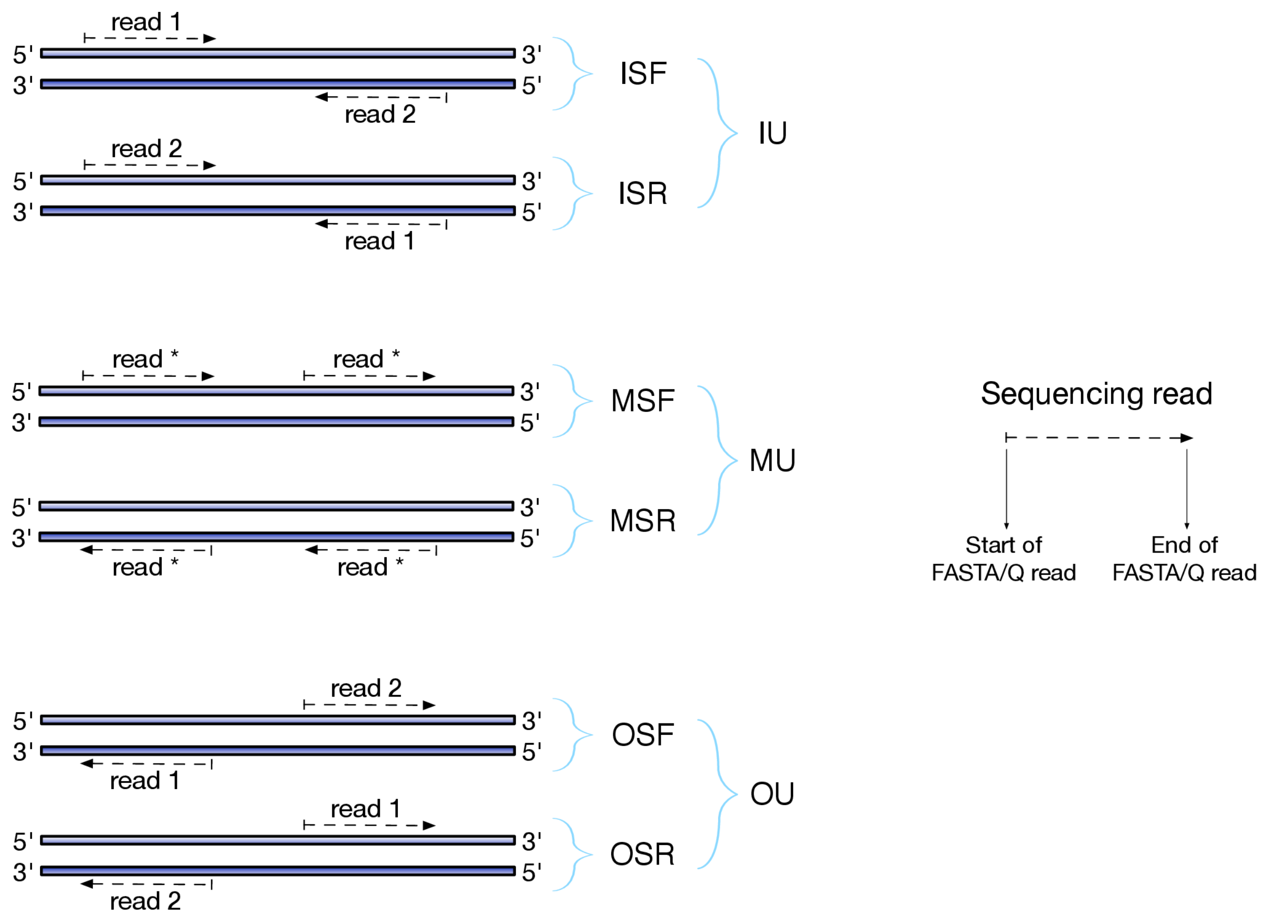
You can control the required number of observations and thus, indirectly, the maximum size of the mapping cache file, via the `-n` argument. Note that the cache itself is considered a "temporary" file, and it is removed from disk by salmon before the program terminates. If you are certain that your read library is large enough that you will observe

the required number of fragments in the first pass, or if you have some other reason to avoid creating the temporary mapping cache, it can disabled with the `--disableMappingCache` flag.

Finally, the purpose of making this beta executable (as well as the Salmon code) available is for people to use it and provide feedback. A pre-print and manuscript are in the works, but the earlier we get feedback, thoughts, suggestions and ideas, the better! So, 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*). Also, please use the same e-mail addresses to contact us with any *detailed* bug-reports (though bug-support for these early beta versions may be slow).

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 unstraned 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**tranted (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 6

Indices and tables

- *genindex*
- *modindex*
- *search*