simpleaf

Release 0.15.0

Dongze He, Rob Patro

Feb 01, 2024

CONTENTS:

CHAPTER

ONE

WHAT IS SIMPLEAF?

`simpleaf` is a program to simplify and customize the running and configuration of single-cell processing with alevin-fry. This documentation covers the main commands of `simpleaf`, and how the program works. The tutorials we have made for simpleaf are available at alevin-fry tutorial website.

CHAPTER

TWO

IMPORTANT NOTE

The simpleaf program runs tools uses in the alevin-fry pipeline. Specifically, to make use of the core functionality of this tool, you will need to install piscem (and / or salmon) and alevin-fry. Further, in order to operate properly, simpleaf **requires that you set the environment variable** ALEVIN_FRY_HOME. It will use the directory pointed to by this variable to cache useful information (e.g. the paths to selected versions of the tools mentioned above, the mappings for custom chemistries you tell it about, and other information like the permit lsits for certain chemistries). So, before you run simpleaf, please make sure that you set the ALEVIN_FRY_HOME environment variable (you can also set it on the command line when you run simpleaf, but setting it in your environment once is much simpler). In most shells, this can be done with

```
$ export ALEVIN_FRY_HOME=/full/path/to/dir/you/want/to/use
```

That's it for initial notes. Use the menu below to learn more about the *simpleaf* commands.

2.1 Installation

Simpleaf can be installed from source, from crates.io, or installed via bioconda. simpleaf alevin-fry, and either piscem or salmon (or both if you prefer), as well as wget.

2.1.1 Recommended: installing from conda

We recommend all x86 (Linux or Mac) users to install simpleaf from bioconda, because all its dependencies are also available on conda, and will be automatically installed (except piscem) when installing simpleaf.

```
conda install simpleaf piscem -c bioconda -c conda-forge
```

For Apple-silicon computers, for example those with an Apple M-series chip, simpleaf should be installed under the x86 emulation layer, in other words, in shell with Rosetta2 enabled. See this for details. Furthermore, if one would like to use piscem on apple silicon, one has to either download the pre-built piscem executable or build piscem from source **in the native shell (with Rosetta2 disabled)** using the commands described here. Then, piscem can be executed from both Rosetta2 enabled and disabled shell.

2.1.2 Installing with cargo

cargo is the rust package manager. `simpleaf` is available on crate.io and can be installed from cargo.

```
cargo install simpleaf
```

Once installed, one will need to set the path to the executable of dependencies using the `simpleaf set-paths` program as discussed in section Set Up Simpleaf manually.

2.1.3 Building from source (from GitHub)

You can also choose to build simpleaf from source by pulling its GitHub repo and build it as a normal rust program. Then, one needs to set up simpleaf manually.

```
git clone https://github.com/COMBINE-lab/simpleaf.git && cd simpleaf
cargo build --release
```

2.2 `set-paths` command

The `set-paths` command is used to set the paths to the relevant executables and store them in a configuration file in the `ALEVIN_FRY_HOME` directory. If you don't provide an explicit path for a program, `simpleaf` will look in your `PATH` for a compatible version. Once paths are set with this command, they are cached in a file in the `ALEVIN_FRY_HOME` directory, and used to execute other commands in `simpleaf`. If you wish to update the paths, you can run this command again, and it will *overwrite* this cache. This command takes the following optional arguments:

```
set paths to the programs that simpleaf will use

Usage: simpleaf set-paths [OPTIONS]

Options:
  -s, --salmon <SALMON>          path to salmon to use
  -p, --piscem <PISCEM>          path to piscem to use
  -a, --alevin-fry <ALEVIN_FRY>  path to alein-fry to use
  -h, --help                     Print help information
  -V, --version                  Print version information
```

2.3 `add-chemistry` command

The `add-chemistry` command simply allows adding a new custom chemistry to `simpleaf`'s registry of recognized chemistries. The sub-command takes a `--name` (a string you wish to use to designate this chemistry), and a `--geometry` (a custom geometry string to which you want to map this chemistry). The usage is as below. Note, if you attempt to add a chemistry for a name that already exists in the custom chemistry registry, the new geometry will overwrite the existing one. For more details on the syntax used to describe custom geometries, see the relevant documentation on the custom chemistry specification.

```
USAGE:
    simpleaf add-chemistry --name <NAME> --geometry <GEOMETRY>

OPTIONS:
```

(continues on next page)

(continued from previous page)

```
    -g, --geometry <GEOMETRY>    the geometry to which the chemistry maps
    -h, --help                   Print help information
    -n, --name <NAME>            the name to give the chemistry
```

2.4 inspect command

The `inspect` command simply inspects the `ALEVIN_FRY_HOME` directory, and prints to the console the relevant information contained therein. Specifically, it will print the paths currently being used for the `piscem`, `salmon`, and `alevin-fry` programs, as well as the versions of these tools. Further, if a `custom_chemistries.json` file is present, it will print the contents of this file. This command can be invoked as:

```
$ simpleaf inspect
```

Currently, this command takes no arguments or options.

2.5 index command

The `index` command has two forms of input; either it will take a reference genome FASTA and GTF as input, from which it can build a spliced+intronic (splici) reference or a spliced+unspliced (spliceu) reference using roers (which is used as a library directly from `simpleaf`, and so need not be installed independently), or it will take a single reference sequence file (i.e. FASTA file) as input (direct-ref mode).

In expanded reference mode, after the expanded reference is constructed, the resulting reference will be indexed with `piscem build` or `salmon index` command (depending on the mapper you choose to use), and a copy of the 3-column transcript-to-gene file will be placed in the index directory for subsequent use. The output directory will contain both a `ref` and `index` subdirectoy, with the first containing the splici reference that was extracted from the provided genome and GTF, and the latter containing the index built on this reference.

In direct-ref mode, the provided fasta file (passed in with `--refseq`) will be provided to `piscem build` or `salmon index` directly. The output diretory will contain an `index` subdirectory that contains the index built on this reference.

The relevant options (which you can obtain by running `simpleaf index -h`) are:

```
build the (expanded) reference index

Usage: simpleaf index [OPTIONS] --output <OUTPUT> <--fasta <FASTA>|--ref-seq <REF_SEQ>>

Options:
  -o, --output <OUTPUT>            path to output directory (will be created if it doesn
↪'t exist)
  -t, --threads <THREADS>          number of threads to use when running [default: 16]
  -k, --kmer-length <KMER_LENGTH>  the value of k to be used to construct the index␣
↪[default: 31]
      --keep-duplicates            keep duplicated identical sequences when constructing␣
↪the index
  -p, --sparse                     if this flag is passed, build the sparse rather than␣
↪dense index for mapping
  -h, --help                       Print help information
  -V, --version                    Print version information
```

(continues on next page)

(continued from previous page)

```
Expanded Reference Options:
      --ref-type <REF_TYPE>    specify whether an expanded reference, spliced+intronic␣
↪(or splici) or spliced+unspliced (or spliceu), should be built [default:␣
↪spliced+intronic]
  -f, --fasta <FASTA>          reference genome to be used for the expanded reference␣
↪construction
  -g, --gtf <GTF>              reference GTF file to be used for the expanded reference␣
↪construction
  -r, --rlen <RLEN>            the target read length the splici index will be built for
      --dedup                  deduplicate identical sequences in roers when building an␣
↪expanded reference  reference
      --spliced <SPLICED>      path to FASTA file with extra spliced sequence to add to␣
↪the index
      --unspliced <UNSPLICED>  path to FASTA file with extra unspliced sequence to add␣
↪to the index

Direct Reference Options:
      --ref-seq <REF_SEQ>  target sequences (provide target sequences directly; avoid␣
↪expanded reference construction)

Piscem Index Options:
      --use-piscem                           use piscem instead of salmon for indexing␣
↪and mapping
  -m, --minimizer-length <MINIMIZER_LENGTH>  the value of m to be used to construct the␣
↪piscem index (must be < k) [default: 19]
```

2.6 `quant` command

The `quant` command takes as input either:

1) the index, reads, and relevant information about the experiment (e.g. the chemistry) OR
2) the directory containing the result of a previous mapping run, and relevant information about the experiemnt (e.g. the chemistry)

and runs all relevant the steps of the `alevin-fry` pipeline. When processing a new dataset from scratch, the first option is the one you are likely interested in (you will provide the `--index`, `--reads1` and `--reads2` arguments). **If multiple read files are provided to the `--reads1` and `--reads2` arguments, those files must be comma (,) separated.**

On the other hand, if you have already performed quantification or have, for some other reason, already mapped the reads to produce a RAD file, you can start the process from the mapped read directory directly using the `--map-dir` argument instead. This latter approach makes it easy to test out different quantification approaches (e.g. different filtering options or UMI resolution strategies).

Note: If you use the unfiltered-permit-list `-u` mode for permit-list generation, and you are using either `10xv2` or `10xv3` chemistry, you can provide the flag by itself, and `simpleaf` will automatically fetch and apply the appropriate uniftered permit list. However, if you are using `-u` with any other chemistry, you must explicitly provide a path to the unfiltered permit list to be used. The `-d`/`--expected-ori` flag allows controlling the like-named option that is passed to the `generate-permit-list` command of `alevin-fry`. This is an "optional" option. If it is not provided explicitly, it is set to "both" (allowing reads aligning in both orientations to pass through), unless the chemistry is set as `10xv2` or `10xv3`, in which case it is set as "fw". Regardless of the chemistry, if the user sets this option explicitly, this choice is respected.

2.6.1 A note on the --chemistry flag

Note: The geometry specification language has changed in simpleaf v0.9.0 and above. This change is to unify the geometry description language between simpleaf and the tools in the backend that actually perform the fragment mapping. Further, the new laguage is more general, capable and exensible, so it will be easier to add more features in the future in a backward compatible manner. However, this means that if you have a custom_chemistries.json file from before simpleaf v0.9.0, you will have to re-create that file with the new chemistries by overwriting them with the custom geometry descriptions in the new format.

The --chemistry option can take either a string describing the specific chemisty, or a string describing the geometry of the barcode, umi and mappable read. For example, the string 10xv2 and 10xv3 will apply the appropriate settings for the 10x chromium v2 and v3 protocols respectively. However, general geometries can be provided as well, in case the chemistry you are trying to use has not been added as a pre-registered option. For example, the instead of providing the --chemistry flag with the string 10xv2, you could instead provide it with the string "1{b[16]u[10]x:}2{r:}", or, instead of providing 10xv3 you could provide "1{b[16]u[12]x:}2{r:}".

The custom format is as follows; you must specify the content of read 1 and read 2 in terms of the barcode, UMI, and mappable read sequence. A specification looks like this:

```
1{b[16]u[12]x:}2{r:}
```

In particular, this is how one would specify the 10x Chromium v3 geometry using the custom syntax. The format string says that the read pair should be interpreted as read 1 1{...} followed by read 2 2{...}. The syntax inside the {} says how the read should be interpreted. Here b[16]u[12]x: means that the first 16 bases constitute the barcode, the next 12 constitute the UMI, and anything that comes after that (if it exists) until the end of read 1 should be discarded (x). For read 2, we have 2{r:}, meaning that we should interpret read 2, in it's full length, as biological sequence.

It is possible to have pieces of geometry repeated, in which case they will be extracted and concatenated together. For example, 1{b[16]u[12]b[4]x:} would mean that we should obtain the barcode by extracting bases 1-16 (1-based indexing) and 29-32 and concatenating them togehter to obtain the full barcode. A

Note: If you use a custom geometry frequently, you can add it to a *json* file custom_chemistries.json in the ALEVIN_FRY_HOME directory. This file simply acts as a key-value store mapping each custom geometry to the name you wish to use for it. For example, putting the contents below into this file would allow you to pass --chemistry flarb to the simpleaf quant command, and it would interpret the reads as having the specified geometry (in this case, the same as the 10xv3 geometry). Multiple custom chemistries can be added by simply adding more entries to this *json* file.

```
{
  "flarb" : "1{b[16]u[12]x:}2{r:}"
}
```

The relevant options (which you can obtain by running simpleaf quant -h) are below:

```
quantify a sample

Usage: simpleaf quant [OPTIONS] --chemistry <CHEMISTRY> --output <OUTPUT> --resolution
↪<RESOLUTION> <--knee|--unfiltered-pl [<UNFILTERED_PL>]|--forced-cells <FORCED_CELLS>|--
↪expect-cells <EXPECT_CELLS>> <--index <INDEX>|--map-dir <MAP_DIR>>

Options:
  -c, --chemistry <CHEMISTRY>  chemistry
```

(continues on next page)

(continued from previous page)

```
  -o, --output <OUTPUT>        output directory
  -t, --threads <THREADS>      number of threads to use when running [default: 16]
  -h, --help                   Print help information
  -V, --version                Print version information

Mapping Options:
  -i, --index <INDEX>            path to index
  -1, --reads1 <READS1>          comma-separated list of paths to read 1 files
  -2, --reads2 <READS2>          comma-separated list of paths to read 2 files
  -s, --use-selective-alignment  use selective-alignment for mapping (instead of␣
↪pseudoalignment with structural constraints)
      --use-piscem               use piscem for mapping (requires that index points to␣
↪the piscem index)
      --map-dir <MAP_DIR>        path to a mapped output directory containing a RAD file␣
↪to skip mapping

Permit List Generation Options:
  -k, --knee                             use knee filtering mode
  -u, --unfiltered-pl [<UNFILTERED_PL>]  use unfiltered permit list
  -f, --forced-cells <FORCED_CELLS>      use forced number of cells
  -x, --explicit-pl <EXPLICIT_PL>        use a filtered, explicit permit list
  -e, --expect-cells <EXPECT_CELLS>      use expected number of cells
  -d, --expected-ori <EXPECTED_ORI>      The expected direction/orientation of␣
↪alignments in the chemistry being processed. If not provided, will default to `fw` for␣
↪10xv2/10xv3, otherwise `both` [possible
                                         values: fw, rc, both]
      --min-reads <MIN_READS>            minimum read count threshold for a cell to be␣
↪retained/processed; only used with --unfiltered-pl [default: 10]

UMI Resolution Options:
  -m, --t2g-map <T2G_MAP>        transcript to gene map
  -r, --resolution <RESOLUTION>  resolution mode [possible values: cr-like, cr-like-em,␣
↪parsimony, parsimony-em, parsimony-gene, parsimony-gene-em]
```

2.7 simpleaf workflow commands

The `simpleaf workflow` program exposes a number of different commands that are responsible for exposing different capabilities. The list of current and valid commands are documented below.

Besides, we also provides a *simpleaf workflow utility* library for developers, which contains useful functions for workflow template development. The simpleaf team has been continously improving existing functions and adding new funtion into this library. The codebase is included in protocol esturary.

2.7.1 simpleaf workflow get

The `simpleaf workflow get` command helps fetch the files of a registered simpleaf workflow to a local directory. One can run the *simpleaf workflow list* command to obtain a list of all available workflows. Please check our tutorial on running an workflow from an published template and developing custom template from scratch

It searches the workflow registry according to the string passed to the `--name` (or `-n`) flag, pack all related files into a folder named by the workflow name plus a `_template` and dump the folder in the directory passed to the `--output` (or `-o`) flag. If invoking local workflows, one can skip this step and provide the workflow template directly to *simpleaf workflow run*. For a registered workflow, one should get the workflow template from `simpleaf workflow get`, fill in the required information, and provide the filled template to *simpleaf workflow run* via the `--template` flag.

If the given name is not a valid workflow name, an error will be returned. At the same time, `simpleaf` will search for workflows with a similar name and list those names in the error message.

In the template folder dumpped by `simpleaf workflow get`, the workflow template is named by the workflow name and ends with *.jsonnet*. There might be other library files or log files in the folder, depending on the specific workflow. For example, to pull the workflow for analyzing CITE-seq data, we can do

```
simpleaf workflow get --name cite-seq-ADT+HTO_10xv2 -o output_dir
```

The the workflow template will be exported to `output_dir/cite-seq-ADT+HTO_10xv2_template/cite-seq-ADT+HTO_10xv2.jsonnet`.

Providing information in workflow configuration file

Usually, a published workflow contains four sections:

1) `fast_config`: For most users, this section is the only section needed to be completed, i.e., replacing all `null` in the section by a meaningful value. The Jsonnet program should be smart enough to generate a valid workflow description JSON from the information provided here.
2) `advanced_config`: This section contains advanced options that are not covered in the `fast_config` section. The behavior of the workflow can be finely tuned by providing information in this section.
3) `external_commands`: This section contains all external shell command records. If you see `TBD` in this section, it means that these fields will be filled by the Jsonnet program automatically.
4) `meta_info`: This section contains the meta info of this workflow. Sometimes the information provided here is used for controling global arguments, for example the output directory and the number of threads used for each invoked command.

For most users, the `fast_config` is the only section needed to instantiate the template. To fill the missing information, one just needs to replace the `null` with a meaningful value. For more details, please check out dedicated tutorial on running workflows from an published template.

Full Usage

The relevant options (which you can obtain by running `simpleaf workflow get -h`) are:

```
Get the workflow template and related files of a registered workflow

Usage: simpleaf workflow get --name <NAME> --output <OUTPUT>

Options:
  -o, --output <OUTPUT>  path to dump the folder containing the workflow related files
```

(continues on next page)

(continued from previous page)

```
  -n, --name <NAME>       name of the queried workflow
  -h, --help              Print help
  -V, --version           Print version
```

2.7.2 simpleaf workflow run

The `simpleaf workflow run` command is designed to run potentially complex single-cell data processing workflows using an instantiated simpleaf workflow template. Please check our tutorial on running an workflow from an published template and developing custom template from scratch

`simpleaf workflow run` exposes one required parameter group (though the options are mutually exclusive):

- `--template` takes the path to a `simpleaf` workflow *template* (i.e. an un-evaluated JSONNET file). One can develop their own templates or grab published templates from the protocol estuary GitHub repository using the API we provide via the the `simpleaf workflow get` command, and fill in required information.
- `--manifest` takes the path to a `simpleaf` workflow *manifest* (i.e. a fully-instantiated JSON file that describes and enumrates all of the commands to be executed, with all relevant parameters fully specified). This manifest could e.g. be the result of a prior execution, or the result of applying the `simpleaf workflow patch` command to a template to produce one or more manifests with desired parameters replaced.

Additionally, the user may pass an `--output` parameter to the `run` invokation, but **only if a template is being instantiated and run**, as the `--output` flag does not make sense in the context of a fully-instantiated manifest.

- `--output` takes the path to the output directory for writing the log files and the results generated by invoking workflow commands. Note that paramater will only have an effect if the corresponding template allows passing `output` as an external variable (all of the templates in the protocol estuary do). Further, if the output directory has already been manually overridden in the template, then `--output` will have no effect and will not be used; in this case a warning to this effect will be printed.

When calling `simpleaf workflow run` using a workflow template, `simpleaf` will first instantiate the template, which is a Jsonnet program, into a workflow manifest in JSON format. Whereas the workflow template provides a "template" for the workflow and functions to handle features like basic logic, the resulting workflow manifest is a simple imperative description of the commands to be executed. To provide the greatest flexibility, the only requirement we set for a `simpleaf` workflow template is that in the workflow manifest its results, the fields representing a command record, either a `simpleaf` command or an external shell command, follow the format described in section *Valid simpleaf workflow manifest format*.

Then, `simpleaf workflow run` will traverse the workflow manifest to collect the `simpleaf` and external shell command records and place them into an execution queue, ordered by their `step` number.

`simpleaf workflow run` also exposes multiple flags for controlling the execution flow when invoking the commands. If none of the flags is set, `simpleaf` will invoke all commands in the execution queue.

- If setting the `--no-execution` flag, `simpleaf` will parse the file passed to the `--template` option, write the manifest and log files, and return without invoking any command.
- If setting the `--start-at` flag with a `step` number, `simpleaf` will ignore all previous steps (commands) and begin the invocation from the commands in the execution queue whose *step* is equal or next to that specific starting `step`.
- If setting the `--resume` flag, `simpleaf` will try to find the log file from a previous run in the provided output folder to decide which `step` to begin with.
- If setting the `--skip-step` flag with a set of comma-separated `step` numbers, `simpleaf` will ignore the commands whose `step` is in those numbers.

Workflow Output

`simpleaf workflow run` writes two log files to the output directory passed to `--output`:

- `simpleaf_workflow_log.json`: This file records the meta and logging information of the workflow execution. For example, the runtime of each executed command and the `step` of the start and terminating command. If `--resume` is set, `simpleaf` will try to find this file in the provided output directory to decide which step(command) to start.
- `workflow_execution_log.json`: This file is a modified version of the workflow manifest JSON discussed above. The only modification is that in this file, the `active` field of the successfully invoked commands (return code 0) becomes *false*.

workflow run: Full Usage

The relevant options (which you can obtain by running `simpleaf workflow run -h`) are:

```
Parse and instantiate a workflow template and invoke the workflow commands, or run an␣
↪instantiated manifest directly

Usage: simpleaf workflow run [OPTIONS] <--manifest <MANIFEST>|--template <TEMPLATE>>

Options:
  -t, --template <TEMPLATE>  path to an instantiated simpleaf workflow template
  -o, --output <OUTPUT>      output directory for log files and the workflow outputs␣
↪that have no explicit output directory
  -m, --manifest <MANIFEST>  path to an instantiated simpleaf workflow template
  -h, --help                 Print help
  -V, --version              Print version

Control Flow:
  -n, --no-execution           return after instantiating the template (JSONNET file)␣
↪into a manifest (JSON foramt) without actually executing the resulting manifest
  -s, --start-at <START_AT>    Start the execution from a specific Step. All previous␣
↪steps will be ignored [default: 1]
  -r, --resume                 resume execution from the termination step of a previous␣
↪run. To use this flag, the output directory must contains the JSON file generated from␣
↪a previous run
      --skip-step <SKIP_STEP>  comma separated integers indicating which steps␣
↪(commands) will be skipped during the execution

Jsonnet:
  -j, --jpaths <JPATHS>  comma separated library search paths passing to internal␣
↪Jsonnet engine as --jpath flags
```

The procedure of parsing a simpleaf workflow template

In `simpleaf workflow`, we use the Jrsonnet library, a rust implementation of Jsonnet, to parse and instantiate the workflow template. Any valid Jsonnet program and JSON file is a valid simpleaf workflow template, as long as it can produce a valid workflow manifest. When calling Jrsonnet, `simpleaf workflow` automatically passes the following built-in arguments in addition to the provided template.

1) The output directory passed to `--output` as the external variable `output`.
2) The workflow utility library from the protocol estuary as the external variable `utils`.
3) The path to the `utils` folder in the protocol estuary in `ALEVIN_FRY_HOME` as an additional library search directory.
4) The paths passed to the `--lib-path` flag, if any, as additional library search directories.

This also means that any custom configuration program can access the `__output` and `__utils` variables in the Jsonnet program using `std.extVar("__output")` and `std.extVar("__utils")`. Note that the path to the parent directory of the file passed to `--template` is an additional library search directory in Jrsonnet by default.

Valid simpleaf workflow manifest format

Although any Jsonnet program or JSON file is a valid input for `simpleaf workflow`, it doesn't mean that all such files can be converted to a valid `simpleaf` workflow manifest. To provide the greatest flexibility, we set only the below requirements for the fields representing a command record — either a `simpleaf` command or an external command, in the simpleaf workflow manifest JSON file (not necessarily the template).

- To ease the parsing process, all fields that represent arguments in an **external** command argument list be provided as strings, i.e., wrapped by quotes (`"value"`), even for integers like the number of threads (for example, `[..., "-t", "16", ...]` for some external command that takes a number of threads via the `-t` parameter).
- **A command record field must contain a `step` and a `program_name` sub-field, where the `step` field represents which step, using an unassigned integer, this command constitutes in the workflow. The `program_name` field represents a valid program in the user's execution environment as a string.**
  - For a simpleaf command, the correct `program_name` is the name of the simpleaf command as a string. For example, for `simpleaf index`, it is `"simpleaf index"` and for `simpleaf quant`, it is `"simpleaf quant"`.
  - For an external command such as `awk`, if the binary is invokable given the user's `PATH` environment variable, it can just be `"awk"`; if not, it must contain a valid full path to the binary, for example, `"/usr/bin/awk"`.
- A command record can also have a *"active"* boolean field, representing if this command is active. Simpleaf will ignore (neither parse nor invoke) all commands that are inactive (*{"active": false}*). For command records missing this field, simpleaf will regard them as active commands.
- If a field records a `simpleaf` command, the name of its sub-fields, except `step` and `program_name`, must be valid simpleaf flags (for example, options like `--fasta`, or `-f` for short, for `simpleaf index` and `--unfiltered-pl` (or `-u`) for `simpleaf quant`). Those option names (sub-field names), together with their values, if any, will be used to call the corresponding simpleaf program. Sub-fields not named by a valid simpleaf flag will trigger an error.
- If a field records an external command, it must contain valid `step` and `program_name` sub-fields as described above. In contrast to `simpleaf` command records, all arguments of an external shell command must be provided in an array, in order, with the name `"arguments"`. `simpleaf workflow` will parse the entries in the array to build the actual command in order. For example, to tell `simpleaf workflow` to invoke the command `ls -l -h .` at step 7, one needs to use the following JSON record:

```
{
    "step": 7,
    "program_name": "ls",
    "active": true,
    "arguments": ["-l", "-h", "."]
}
```

2.7.3 simpleaf workflow list

`simpleaf workflow list` lists all workflows in the registry. If one would like to refresh the registry to keep all workflows up to date, please run the *simpleaf workflow refresh* command.

Full Usage

The relevant options (which you can obtain by running `simpleaf workflow list -h`) are:

```
Print a summary of the currently available workflows

Usage: simpleaf workflow list

Options:
-h, --help     Print help
-V, --version  Print version
```

2.7.4 simpleaf workflow patch

The `simpleaf workflow patch` command allow "patching" `simpleaf` workflows. Specifically, it allows one to patch either a *workflow template* prior to instantiation (and therefore, to patch the values of variables in the workflow that may affect large parts of the configuration) or a *workflow manifest* (where patching only directly affects the specific fields being replaced). Here, the act of patching refers to the replacement of one or more fields in the template or manifest with alternative values drawn from a parameter table (e.g. a "sample sheet").

The patch command is useful when you wish to use the "skeleton" of a workflow (e.g. a template with many of the variables set), but you wish to parameterize other fields over some set of different options.

Concretely, for example, you may have many gene 10x chromium v3 samples, all of which you wish to process with the same workflow, but providing different reads as input and different output locations for the workflow output. The `patch` command makes this easy to accomplish.

When operating on a template, the patch command takes as input a workflow template (which can be uninstantiated or partially filled in) via the `--template` parameter, as well as a parameter table as a `;` separated CSV file via the `--patch` parameter (see details on the format *below*). For each (non-header) row in the CSV file, it will patch the template with parameters provided in this row, instantiate a new manifest from this template (*after* replacement), and write the instantiated manifest out to a `JSON` file.

When operating on a manifest, the patch command takes as input a manifest files via the `--manifest` parameter, as well as a parameter table as a `;` separated CSV file via the `--patch` parameter. For each (non-header) row in the CSV file, it will patch the manifest with parameters provided in this row and write the resulting patched manifest out to a `JSON` file. Note that, in this case, since the input being patched is a fully-instantiated manifest, the patch simply replaces the values of the designated fields, but it will not affect the values of any fields that are not diretly patched.

Finally, the `workflow patch` command accepts an *optional* argument `--output`, which specifies the path to a directory (which will be recursively created if it doesn't exist) where the patched manifests will be written. If this argument is not provided, then the patched manifests will be written to the same directory as the template or manifest to which the patching process is being applied. The name of each patched manifest is determined by the name of the template or manifest being patched, and the corresponding value in the `name` column of the patch file. For example if you are patching a template named `awesome_gene_expression.jsonnet`, and you have a parameter table with two (non-header) rows with the values in the name column being `sample1` and `sample2` respectively, then the patching process will result in two manifests named `awesome_gene_expression_sample1.json` and `awesome_gene_expression_sample2.json`. If no `--output` was provided to the command, then these will be written in the same directory where `awesome_gene_expression.jsonnet` resides, otherwise they will be written in the directory specified by the `--output` option.

Patch file

The patch file should be a `;`-separated CSV-like file. The header column will contain one entry for each field of the template or manifest that is to be replaced, as well as an additional special column called `name`, that gives a name to the parameter tuple encoded in each row. To refer to a field in the template or manifest, one should use the JSON pointer syntax described in RFC6901. Additionally, since the values being replaced can be of distinct valid JSON types, this type information must also be encoded in the column header.

For example, imagine that your template has two fields defined as below:

```
{
   "workflow" : {
     /* possibly other content */
     "simpelaf_quant" : {
       /* possibly other content */
       "--reads1" : "reads0_1.fq.gz",
        /* possibly other content */
       "ready" : false,
     }
     /* possibly other content */
   }
}
```

that you wish to replace. You wish to replace "–reads1" with "reads1_sample2_1.fq.gz" and "ready" with `true` (the boolean value true, not the string). Then the definition of the corresponding column headers would be `/workflow/simpleaf_quant/--reads1` and `<b>/workflow/simpleaf_quant/ready`, respectively. The <b> before the second column header designates that this column will hold boolean parameters. You could also prefix the first column header with <s> (for the string type), but this is the default and can be omitted. Finally, then, the full patch file might look something like:

```
name;/workflow/simpleaf_quant/--reads1;<b>/workflow/simpleaf_quant/ready
sample1;"reads1_sample2_1.fq.gz";true
```

The valid type tags are:

<s>
prefixes a header that is a pointer to a string-valued field. This is also the default so if a header has no prefix, then <s> is implicitly assumed

<b>
prefixes a header that is a pointer to a boolean-valued field.

<a>
prefixes a header that is a pointer to an array-valued field.

Note: Currently, patching does not support parameter entries that are themselves full object types. Finally, if any entry contains the string "null", it will automatically be converted into the JSON `null` constant (which means that currently, at least, there is a restriction that the string "null" can not be patched into a template or manifest).

workflow patch: Full Usage

The relevant options (which you can obtain by running `simpleaf workflow patch -h`) are:

```
Patch a workflow template or instantiated manifest with a subset of parameters to␣
↪produce a series of workflow manifests

Usage: simpleaf workflow patch [OPTIONS] --patch <PATCH> <--manifest <MANIFEST>|--
↪template <TEMPLATE>>

Options:
  -m, --manifest <MANIFEST>  fully-instantiated manifest (JSON file) to patch. If this␣
↪argument is given, the patch is applied directly
                             to the JSON file in a manner akin to simple key-value␣
↪replacement. Since the manifest is fully-instantiated,
                             no derived values will be affected
  -t, --template <TEMPLATE>  partially-instantiated template (JSONNET file) to patch. If␣
↪this argument is given, the patch is applied
                             *before* the template is instantiated (i.e. if you override␣
↪a variable used elswhere in the template, all
                             derived values will be affected)
  -p, --patch <PATCH>        patch to apply as a ';' separated parameter table with␣
↪headers declared as specified in the documentation
  -o, --output <OUTPUT>      output directory where the patched manifest files (i.e. the␣
↪output of applying the patching procedure)
                             should be stored. If no directory is provided, the patched␣
↪manifests are stored in the same location as the
                             input template or manifest to which patching is applied
  -h, --help                 Print help
  -V, --version              Print version
```

2.7.5 simpleaf workflow refresh

`simpleaf workflow refresh` pulls the latest protocol estuary library from its GitHub repository. We recommend updating the registry everytime before fetching a workflow.

Full Usage

The relevant options (which you can obtain by running `simpleaf workflow refresh -h`) are:

```
Update the local copy of protocol esturary to the latest version

Usage: simpleaf workflow refresh

Options:
-h, --help     Print help
-V, --version  Print version
```

2.7.6 Simpleaf workflow utility library

To ease the development of *simpleaf* workflow templates, the *simpleaf* team provides not only some built-in variables, but also a workflow utility library, which will be automatically passed to the internal Jsonnet engine of *simpleaf* when parsing a workflow template as the `__utils` external variable. One can receive this variable in their templates by adding `utils=std.extVar("__utils")`, and use the functions in the utility library by calling `utils.function_name(args)`, where *function_name* should be replaced by an actual function name listed below. To be consistent with the Jsonnet official documentation, here we will list the function signatures with a brief description. One can find the function definitions on this page.

Import the utility library

As the built-in variables are provided by *simpleaf* to its internal Jsonnet engine, they will be unavailable if we want to parse the template directly using Jsonnet or Jrsonnet. Therefore, when debugging templates that utilize the utility library with an external Jsonnet engine, we must manually provide the `__utils` external variable to Jsonnet and either copy and paste the library file to the same directory as the template file, which is the default library searching path when calling jsonnet, or provide the directory containing the library as an additional library searching path. Although in the following code chunk, we show the code for both ways, we only need to select one in practice. Here we assume that *simpleaf* has been configured correctly, i.e., the `ALEVIN_FRY_HOME` environment variable has been set and a local copy of the protocol-estuary exists. If not, one can directly obtain the library file from its GitHub repository.

```
# If we haven't set up simpleaf, we pull the file directly from github
wget https://github.com/COMBINE-lab/protocol-estuary/blob/main/utils/simpleaf_workflow_
↪utils.libsonnet

# Otherwise, we either copy the library file to the same dir as the template
copy $ALEVIN_FRY_HOME/protocol-estuary/protocol-estuary-main/utils/simpleaf_workflow_
↪utils.libsonnet .

# or provide the directory as an additional library searching path via --jpath
jsonnet a_template_using_utils_lib.jsonnet --ext-code '__utils=import "simpleaf_workflow_
↪utils.libsonnet"' --jpath "$ALEVIN_FRY_HOME/protocol-estuary/protocol-estuary-main/
↪utils"
```

where `--ext-code` is the flag for passing an external variable, and `--jpath` specifies the library searching path.

Although *simpleaf* automatically provides the utility library as the external variable `__utils`, we must receive this external variable in our template before starting using the functions provided in this library.

To do this, we recommend adding the following code at the beginning of your workflow template.

```
local utils=std.extVar("__utils");
```

utils.ref_type(o)

Input:

- **o: an object with**
  - a *type* field and
  - an object field with the name specified by the *type* field. Other fields will be ignored. For example, `{type: "spliceu", spliceu: {gtf: "genes.gtf", fasta: "genome.fa", "field_being_ignored": "ignore me"}}`.

Output:

- An object with the *simpleaf index* arguments that are related to the specified reference type in the input object.

This function has four modes (reference types), triggered by the `type` field in the input object. When specifying a mode, the input object must contain an object field named by that mode and contain the required fields. Otherwise, an error will be raised. The four modes are:

- ***spliceu* (*spliced+unspliced* reference): The required fields are:**
  - `gtf`: A string representing the path to a gene annotation GTF file.
  - `fasta`: A string representing the path to a reference genome FASTA file.
- ***splici* (*spliced+intronic* reference): The required fields are:**
  - `gtf`: A string representing the path to a gene annotation GTF file.
  - `fasta`: A string representing the path to a reference genome FASTA file.
  - `rlen`: An *optional* field representing the read length in the dataset. If not provided, the default value, 91, will be used.
- ***direct_ref*: The required fields are:**
  - `ref_seq`: A string representing the path to a *transcriptome* FASTA file.
  - `t2g_map`: A string representing the path to a transcript-to-gene mapping file.
- ***existing_index*: The required fields are:**
  - `index`: A string representing the path to an existing index directory.
  - `t2g_map`: A string representing the path to a transcript-to-gene mapping file.

Wrapper functions: We also provide separate functions for each of the four modes, `utils.splici`, `utils.spliceu`, `utils.direct_ref`, and `utils.existing_index`, which are thin wrappers of `utils.ref_type`. These four functions take an object containing their required fields introduced above.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local splici_args = {
    gtf : "genes.gtf",
    fasta : "genome.fa",
    rlen : 91
};

local ref_type = utils.ref_type({
    type : "splici",
    splici : splici_args
});

local splici = utils.splici(splici_args);
```

In the above example, the objects `ref_type` and `splici` are identical and look like the following:

```
{
    # hidden, system fields
    type :: "splici", # hidden field
```

(continues on next page)

(continued from previous page)

```
    arguments :: {gtf : "genes.gtf", fasta : "genome.fa", rlen : 91}, # hidden field

    # fields shown in the manifest
    "--ref-type" : "splici",
    "--fasta" : "genome.fa",
    "--gtf" : "genes.gtf",
    "--rlen" : 91
}
```

utils.simpleaf_index(step, ref_type, arguments, output)

Input:

- `step`: An integer indicating the step number (execution order) of this simpleaf command record in the workflow.
- `ref_type`: A `ref_type` object returned by calling `utils.ref_type` or any object with the same format.
- `arguments`: An object in which each field represents a *simpleaf index* argument. Furthermore, there must be a field called `active` representing the active state of this *simpleaf index* command.
- `output`: A string representing the output directory of the *simpleaf index* command.

Output:

- A well-defined *simpleaf index* command record.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local splici_args = {
    gtf : "genes.gtf",
    fasta : "genome.fa",
    rlen : 91
};

local splici = utils.splici(splici_args);

local arguments = {
    active : true,
    "--use-piscem" : true
};

local simpleaf_index = utils.simpleaf_index(
    1, # step number
    splici, # ref_type,
    arguments,
    "./simpleaf_index" # output directory
);
```

The `simpleaf_index` object in the above code chunk will be

```
{
    # hidden, system fields
```

(continues on next page)

(continued from previous page)

```
    ref_type :: {}, # hidden field. The actual contents are omitted. see above example␣
↪code for function `ref_type`
    arguments :: {active : true, "--use-piscem" : true},  # hidden field
    output :: "./simpleaf_index", # hidden field
    index :: "./simpleaf_index/index", # hidden field
    t2g_map :: "./simpleaf_index/index/t2g_3col.tsv", # hidden field

    # fields shown in in the manifest
    program_name : "simpleaf index",
    step : 1,
    active : true,
    "--output": "./workflow_output/simpleaf_index",
    "--gtf" : "genes.gtf",
    "--fasta" : "genome.fa",
    "--rlen" : 91,
    "--use-piscem" : true
}
```

utils.map_type(o, simpleaf_index = {})

Input:

- **`o`: an object with**
  - a `type` field, and
  - an object field with the name specified by the `type` field. Other fields will be ignored. For example, `{"type": "map_reads", "map_reads": {"reads1": null, "reads2": null}, "field_being_ignored": "ignore me"}`.
- `simpleaf_index`: An empty object if in *existing_mappings* mode, or the output object of the *simpleaf_index* function if in *map_reads* mode. The default value is an empty object.

Output:

- An object with the *simpleaf quant* arguments that are related to the specified map type in the input object.

This function has two modes (map types), triggered by the *type* field in the input object. When specifying a mode, the input object must contain an object field named by that mode and contain the required fields. Otherwise, an error will be raised. The two modes are:

- ***map_reads*: Map reads against the provided index or an index built from a previous step. The required fields are**
  - `reads1`: A string representing the path to a gene annotation GTF file,
  - `reads2`: A string representing the path to a reference genome FASTA file.
- ***existing_mappings*: Skip mapping and use the existing mapping results. The required fields are**
  - `map_dir`: A string representing the path to the mapping result directory,
  - `t2g_map`: A string representing the path to a transcript-to-gene mapping file.

Wrapper functions: We also provide separate functions for each of the two modes, `utils.map_reads` and `utils.existing_mappings`, which are thin wrappers of `utils.map_type`. These two functions take an object containing their required fields introduced above.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local simpleaf_index = {}; # The return of object of simpleaf_index function in its␣
↪example usage

local map_reads_args = {
    reads1 : "reads1.fastq",
    reads2 : "reads2.fastq"
};

local map_type = utils.map_type({
    type : "map_reads",
    map_reads : map_reads_args
});

local map_reads = utils.map_reads(map_reads_args);
```

In the above example, the objects `map_type` and `map_reads` are identical and look like the following:

```
{
    # hidden, system fields
    type :: "map_reads", # hidden field
    arguments :: {reads1 : "reads1.fastq", reads2 : "reads2.fastq"}, # hidden field

    # fields shown in the manifest
    "--index" : "./workflow_output/simpleaf_index/index",
    "--t2g-map": "./workflow_output/simpleaf_index/index/t2g_3col.tsv",
    "--reads1" : "reads1.fastq",
    "--reads2" : "reads2.fastq"
}
```

utils.cell_filt_type(o)

Input:

- **o: an object with**
  - a `type` field, and
  - an argument field with the name specified by the `type` field. Other fields will be ignored. For example, `{"type": "explicit_pl", "explicit_pl": "whitelist.txt", "field_being_ignored": "ignore me"}`

Output:

- An object with the *simpleaf quant* arguments that are related to the specified cell filtering type in the input object.

This function has five modes (cell filtering types), triggered by the *type* field in the input object. When specifying a mode, the input object must contain an object field named by that mode and contain the required fields. Otherwise, an error will be raised. For more details, please refer to the online documentation of simpleaf quant and alevin-fry. The five modes are:

- *unfiltered_pl*: No cell filtering but correcting cell barcodes by an external or default (only works for 10X Chromium V2 and V3). The corresponding argument value field can be `true` (using the default whitelist if in *10xv2* and *10xv3* chemistry), or a string representing the path to an unfiltered permit list file.

- *knee*: Knee point-based filtering. The corresponding argument value field must be *true* if selected.
- *forced*: Use a forced number of cells. The corresponding argument field must be an integer representing the number of cells that can pass the filtering.
- *expect*: Use the expected number of cells. The corresponding argument field must be an integer representing the expected number of cells.
- *explicit_pl*: Use a filtered, explicit permit list. The corresponding argument field must be a string representing the path to a cell barcode permit list file.

Wrapper functions: We also provide a separate function for each mode, `utils.unfiltered_pl`, `utils.knee`, `utils.forced`, `utils.expect`, and `utils.explicit_pl`, which are thin wrappers of `utils.cell_filt_type`. These functions take an object containing their required fields introduced above.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local unfiltered_pl_args = {
    unfiltered_pl : true
};

local cell_filt_type = utils.cell_filt_type({
    type : "unfiltered_pl",
    unfiltered_pl : unfiltered_pl_args
});

local unfiltered_pl = utils.unfiltered_pl(unfiltered_pl_args);
```

In the above example, the objects *cell_filt_type* and *unfiltered_pl* are identical and look like the following:

```
{
    # hidden, system fields
    type :: "unfiltered_pl", # hidden field
    arguments :: true, # hidden field

    # fields shown in the manifest
    "--unfiltered-pl" : true
}
```

simpleaf_quant(step, map_type, cell_filt_type, arguments, output)

Input:

- `step` : An integer indicating the step number (execution order) of this simpleaf command record in the workflow.
- `map_type` : A *map_type* object returned by calling *utils.map_type* or any object with the same format.
- `cell_filt_type` : A *cell_filt_type* object returned by calling *utils.cell_filt_type* or any object with the same format.
- `arguments` : an object in which each field represents a *simpleaf quant* argument. Furthermore, there must be a field called `active` representing the active state of this simpleaf index command.
- `output` : A string representing the output directory of this *simpleaf quant* command.

Output:

- A well-defined *simpleaf quant* command record.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local arguments = {
    active : true,
    "--chemistry" : "10xv3",
    "--resolution" : "cr-like"
};

local simpleaf_quant = utils.simpleaf_quant(
    2, # step number
    map_type, # defined in the example usage of function `map_reads`
    cell_filt_type, # defined in the example usage of function `cell_filt_type`
    arguments,
    "./simpleaf_quant" # output directory
);
```

The `simpleaf_quant` object in the above code chunk will be

```
{
    # hidden, system fields
    map_type :: {}, # hidden field. The actual contents are omitted. see above example␣
↪code for function `map_reads`
    cell_filt_type :: {}, # hidden field. The actual contents are omitted. see above␣
↪example code for function `cell_filt_type`
    arguments :: {active : true, "--chemistry" : "10xv3", "--resolution" : "cr-like"},
↪# hidden field
    output :: "./simpleaf_quant", # hidden field

    # fields shown in in the manifest
    program_name : "simpleaf index",
    step : 1,
    active : true,
    "--chemistry": "10xv3",
    "--index": "./workflow_output/simpleaf_index/index",
    "--min-reads": 10,
    "--output": "./workflow_output/simpleaf_quant",
    "--reads1": "reads1.fastq",
    "--reads2": "reads2.fastq",
    "--resolution": "cr-like",
    "--t2g-map": "./workflow_output/simpleaf_index/index/t2g_3col.tsv",
    "--unfiltered-pl": true
}
```

feature_barcode_ref(start_step, csv, name_col, barcode_col, output)

Input:

- `start_step`: An integer indicating the starting step number (execution order) of the series of command records in the workflow. This function will define three command records with incremental step numbers according to the provided step number.
- `csv`: A string representing the path to the "feature_barcode.csv" file of the dataset.
- `name_col`: An integer representing the column index of the feature name column in the feature barcode CSV file.
- `barcode_col`: An integer representing the column index of the feature barcode sequence column in the feature barcode CSV file.
- `output`: A string representing the parent output directory of the result files. It will be created if it doesn't exist.

Output:

- An object containing three external command records, including *mkdir*, *create_t2g*, and *create_fasta*, and a hidden object that follows the output format of *utils.ref_type* shown above. This *ref_type* object is of the *direct_ref* type. It can be used as the second argument of *utils.simpleaf_index*. In this *ref_type* object,

This function defines three external command records:

1. *mkdir*: This command calls the *mkdir* shell program to create the output directory recursively if it doesn't exist.
2. *create_t2g*: This command calls *awk* to create a transcript-to-gene mapping TSV file according to the input *csv* file, in which the transcript ID and gene ID of each feature barcode are identical. The expected output file of this command will be named ".feature_barcode_ref_t2g.tsv" and located in the provided output directory.
3. *create_fasta*: This command calls *awk* to create a FASTA file according to the input *csv* file, in which each feature barcode is a FASTA record. The expected output file of this command will be named ".feature_barcode_ref.fa" and located in the provided output directory.

Please note that the *start_step* argument represents the starting step of the series of external commands. If *start_step* is set to 1, then *mkdir* will be assigned step 1, *create_t2g* will be assigned step 2, and so on. Therefore, the step of any future command after the *utils.feature_barcode_ref* commands should not be less than 4.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");

local feature_barcode_ref = utils.feature_barcode_ref(
    1, # start step number
    "feature_barcode.csv", # feature barcode csv
    1, # name_column
    5, # barcode column
    "feature_barcode_ref" # output path
)
```

The resulting object will look like the following:

```
{
    # hidden, system fields
    step :: 1,
    last_step :: 3,
    csv :: "feature_barcode.csv",
```

(continues on next page)

(continued from previous page)

```
    output :: "./feature_barcode_ref",
    ref_seq :: "./feature_barcode_ref/.feature_barcode_ref.fa",
    t2g_map :: "./feature_barcode_ref/.feature_barcode_ref_t2g.tsv",

    # external command records
    mkdir : {
        active : true,
        step: step,
        program_name: "mkdir",
        arguments: ["-p", "./feature_barcode_ref"]
    },
    create_t2g : {
        active : true,
        step: step + 1,
        program_name: "awk",
        arguments: ["-F","','","'NR>1 {sub(/ /,\"_\",$1);print $1\"\\t\"$1}'", csv, ">",
↪"./feature_barcode_ref/.feature_barcode_ref_t2g.tsv"]
    },

    create_fasta : {
        active : true,
        step: step + 2,
        program_name: "awk",
        arguments: ["-F","','","'NR>1 {sub(/ /,\"_\",$1);print \">\"$1\"\\n\"$5}'", csv,
↪">", "./feature_barcode_ref/.feature_barcode_ref.fa"]
    },
    ref_type :: {
        type : "direct_ref",
        t2g_map :: "./feature_barcode_ref/.feature_barcode_ref_t2g.tsv",
        "--ref-seq" : "./feature_barcode_ref/.feature_barcode_ref.fa"
    }
}
```

barcode_translation(start_step, url, quant_cb, output)

Input:

- `start_step`: An integer indicating the starting step number (execution order) of the series of command records in the workflow. This function will define five command records with incremental step numbers according to the provided step number.
- `url`: A string representing the downloadable URL to the barcode mapping file. You can use this URL for 10xv3 data.
- `quant_cb`: A string representing the path to the cell barcode file. Usually, this is at *af_quant/alevin/quants_mat_rows.txt* in the simpleaf quant command output directory.
- `output`: A string representing the parent output directory of the result files. It will be created if it doesn't exist.

Output:

- An object containing five external command records, including *mkdir*, *fetch_cb_translation_file*, *unzip_cb_translation_file*, *backup_bc_file*, and *barcode_translation*.

This function defines five external command records:

1. *mkdir*: This command calls the *mkdir* shell program to create the output directory recursively if it doesn't exist.
2. *fetch_cb_translation_file*: This command calls *wget* to fetch the barcode mapping file. The expected output file of this command will be called ".barcode.txt.gz", located in the provided output directory.
3. *unzip_cb_translation_file*: This command calls *gunzip* to decompress the barcode mapping file. The expected output file of this command will be called ".barcode.txt", located in the provided output directory.
4. *backup_bc_file*: This command calls *mv* to rename the provided barcode file. The expected output file of this command will have the same path as the provided barcode file but with a *.bkp* suffix.
5. *barcode_translation*: This command calls *awk* to convert the barcodes in the provided barcode file according to the barcode translation file. The expected output file will be put at the provided *quant_cb* path.

Notice that the *start_step* argument represents the starting step of the series of external commands. If *start_step* is set to 1, then *mkdir* will be assigned as step 1, *fetch_cb_translation_file* will be assigned step 2, and so on. Therefore, the step of any future command after the *barcode_translation* commands should not be less than 6.

Example Usage

```
# import the utility library
local utils=std.extVar("__utils");
local url = "https://github.com/10XGenomics/cellranger/raw/master/lib/python/cellranger/
↪barcodes/translation/3M-february-2018.txt.gz";
local quant_cb = "simpeaf_quant/af_quant/alevin/quants_mat_rows.txt";

local barcode_translation = utils.barcode_translation(
    1, # start step number
    url,
    quant_cb,
    "simpeaf_quant/af_quant/alevin" # output path
)
```

The resulting object will look like the following:

```
{
    step :: 1,
    last_step :: 5,
    url :: "https://github.com/10XGenomics/cellranger/raw/master/lib/python/cellranger/
↪barcodes/translation/3M-february-2018.txt.gz",
    quant_cb :: "simpeaf_quant/af_quant/alevin/quants_mat_rows.txt",
    output :: "simpeaf_quant/af_quant/alevin",
    mkdir : {
        active : true,
        step : step,
        program_name : "mkdir",
        arguments : ["-p", "simpeaf_quant/af_quant/alevin"]
    },

    fetch_cb_translation_file : {
        active : true,
        step : step + 1,
        program_name : "wget",
        arguments : ["-O", "simpeaf_quant/af_quant/alevin/.barcode.txt.gz", "https://
↪github.com/10XGenomics/cellranger/raw/master/lib/python/cellranger/barcodes/
↪translation/3M-february-2018.txt.gz"]
    },
```

(continues on next page)

(continued from previous page)

```
    unzip_cb_translation_file : {
        active : true,
        step : step + 2,
        "program_name" : "gunzip",
        "arguments": ["-c", "simpeaf_quant/af_quant/alevin/.barcode.txt.gz", ">",
↪"simpeaf_quant/af_quant/alevin/.barcode.txt"]
    },

    backup_bc_file : {
        active : true,
        step: step + 3,
        program_name: "mv",
        arguments: ["simpeaf_quant/af_quant/alevin/quants_mat_rows.txt", "simpeaf_quant/
↪af_quant/alevin/quants_mat_rows.txt.bkp"]
    },

    // Translate RNA barcode to feature barcode
    barcode_translation : {
        active : true,
        step: step + 4,
        program_name: "awk",
        arguments: ["'FNR==NR {dict[$1]=$2; next} {$1=($1 in dict) ? dict[$1] : $1}1'",
↪"simpeaf_quant/af_quant/alevin/.barcode.txt", "simpeaf_quant/af_quant/alevin/quants_
↪mat_rows.txt.bkp", ">", "simpeaf_quant/af_quant/alevin/quants_mat_rows.txt"]
    },
}
```

utils.get(o, f, use_default = false, default = null)

Input:

- o: an object,
- f: the target field name,
- use_default: boolean,
- default: a default value returned if the target field doesn't exist.

Output:

- Return the target field *f* in the given object if the object has a sub-field called f. Otherwise,
- if use_default is true, return the value of the default argument.
- if use_default is false, raise an error.

This function tries to (non-recursively) get a sub-field in the provided object and return it. If the field doesn't exist, then it either returns a default value or raises an error.

Example Usage

```
local utils = std.extVar("__utils");

local splici_args = {
```

(continues on next page)

(continued from previous page)

```
    gtf : "genes.gtf",
    fasta : "genome.fa",
    rlen : 91
};

{
    default_behavior : utils.get(splici_args, "gtf"), # this will return "genes.gtf",

    not_exist : utils.get(splici_args, "I do not exist"), # raise error

    provide_default : utils.get(splici_args, "I do not exist", true, "but I have a␣
↪default value") # this yields "but I have a default value"

}
```

2.8 License

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search