BioLite Documentation

Release 0.3.0

Mark Howison, Casey Dunn, Nick Sinnott-Armstrong

February 08, 2013

CONTENTS

BioLite is a Python/C++ framework for implementing bioinformatics pipelines for Next-Generation Sequencing (NGS) data, in particular pair-end Illumina data.

BioLite is designed around three priorities:

- automating the collection and reporting of *diagnostics*;
- tracking *provenance* of analyses;
- and providing lightweight tools for building out customized analysis *pipelines*.

Where possible, we have wrapped existing bioinformatics tools, especially for assembly, alignment and annotation. For analyses where a tool does not exist or is not optimized for the high computational and storage requirements of NGS data, we have developed custom tools in C++ after the standard UNIX "pipe and filter" design pattern.

Our primary motivation for developing BioLite is to implement Agalma, a *de novo* transcriptome assembly and annotation pipeline for Illumina data.

CHAPTER

ONE

CONTENTS

1.1 Installation

For quick installation instructions for OS X and Ubuntu, see the README. This file has more detailed instructions for installation on other platforms or for developers.

After installation, proceed to the configuration instructions at the end of this document.

If you would like to install to a location other than /usr/local (if you don't have permission to write to /usr/local, for example), see the "Installing to an alternative location" section.

1.1.1 Prerequisites

This section lists required and optional prerequisites, with notes on specific versions we have tested and found to work.

To compile and install BioLite, you must at a minimum have:

- A C/C++ compiler that supports OpenMP and the TR1 standard. Tested:
  - gcc 4.4.6 (CentOS 6.3)
  - gcc 4.6.3 (Ubuntu 12.04)
  - XCode gcc 4.2.1 (OS X 10.8)
- Python (2.7.2, 2.7.3) with packages:
  - biopython (1.60, 1.61)
  - dendropy (3.12.0)
  - docutils (0.9.1)
  - matplotlib (1.1.0, 1.1.1rc, 1.1.1)
  - networkx (1.6, 1.7)
  - numpy (1.6.1, 1.6.2)

BioLite provides a large collection of wrappers for the following 3rd party bioinformatics tools. While you do not have to install these to be able to load the BioLite python library or to use the BioLite command-line tools, a BioLite script that calls a wrapper must be able to find the corresponding program in your PATH. BioLite comes with a shell script to automate downloading and building many of these 3rd party programs. See the "Installing 3rd Party Software" section below for more details.

- FastQC 0.10.0 (http://www.bioinformatics.babraham.ac.uk/projects/fastqc/)
- Blast+ 2.2.27 (http://blast.ncbi.nlm.nih.gov/)

- Bowtie 0.12.8 (http://bowtie-bio.sourceforge.net/)
- Bowtie2 2.0.6 (http://bowtie-bio.sourceforge.net/bowtie2/)
- samtools 0.1.18 (http://samtools.sourceforge.net/)
- Velvet 1.2.08 (http://www.ebi.ac.uk/~zerbino/velvet/)

  Note: with LONGSEQUENCES=1 and MAXKMERLENGTH >= 61, recommended 63
- Oases 0.2.08 (http://www.ebi.ac.uk/~zerbino/oases/)

  Note: with LONGSEQUENCES=1 and MAXKMERLENGTH >= 61, recommended 63
- Trinity r2012-10-05 (http://trinityrnaseq.sourceforge.net/)
- MACSE 0.9b1 (http://mbb.univ-montp2.fr/macse/)

  Note: if you install manually, make sure the MACSE jar file is in your PATH.
- RAxML 7.2.8-ALPHA (http://sco.h-its.org/exelixis/software.html)
- Gblocks 0.91b (http://molevol.cmima.csic.es/castresana/Gblocks.html)
- mcl 12-135 (http://micans.org/mcl)

1.1.2 Generic instructions for installing from the tar ball

Unpack the tarball (from https://bitbucket.org/caseywdunn/biolite/downloads):

```
tar xf biolite-X.X.X.tar.gz
cd biolite-X.X.X
```

Build and install 3rd party tools:

```
sudo ./build_3rd_party.sh /usr/local
```

Build and install BioLite:

    ./confgure make sudo make install

Proceed to Configuration at the end of this document.

1.1.3 Installing from the git repo

(Skip this unless you are building a development version that you cloned from Bitbucket.)

First, create a fork of the repository so that you can later send a pull request so that we can incorporate your improvements. Then clone your forked repository.

Build and install 3rd party tools:

```
sudo ./build_3rd_party.sh /usr/local
```

See the Prerequisites section above for other software you may need to install.

You will need to have the automake, autoconf and libtool packages installed. To produce the configure scripts, use the command:

```
./autogen.sh
```

Then you can use the typical build sequence:

```
./configure
make
sudo make install
```

Proceed to Configuration at the end of this document.

1.1.4 Installing to an alternative location

The instructions above assume that you are installing BioLite to /usr/local, which requires root access. If you are installing to another location, modify the installation instructions as follows. These are not full instructions, they just explain how to modify the instructions above.

In the instructions below, we use [installation_path] as a placeholder for the path that you would like to install to, eg /home/lucy/local. Note that [installation_path] needs to be an absolute path.

Modify the command to build and install 3rd party tools to specify the alternative path:

```
./build_3rd_party.sh [installation_path]
```

Modify the configure command to both change the install location and tell it where third party packages were installed:

```
./configure --prefix=[installation_path]
make
make install
```

If you are installing somewhere that you have write access to, you don't need to use sudo for *build_3rd_party.sh* or *make install*.

Proceed to the next section for instructions on setting paths.

1.1.5 Setting PATH and PYTHONPATH

If you install to a canonical location on your system, like '/usr/local', the scripts and programs will already be in your PATH and the python module will be ready to import.

Otherwise, if you want to be able to call the programs without specifying their full path, you need to add the new 'bin' directory to your PATH. In bash (you can add add this to ~/.bashrc):

```
export PATH=[installation_path]/bin:$PATH
```

or in csh:

```
setenv PATH [installation_path]/bin:$PATH
```

To be able to import BioLite's python modules in python, you will also need to add the full path to you PYTHONPATH, replacing the python version below with your version of python (most likely "2.7"). In bash:

```
export PYTHONPATH=[installation_path]/lib/python2.7/site-packages:$PYTHONPATH
```

or in csh:

```
setenv PYTHONPATH [installation_path]/lib/python2.7/site-packages:$PYTHONPATH
```

1.1.6 Installing 3rd Party Software

The BioLite source comes with a shell script that will download the required 3rd party software from a mirror at Brown's Center for Computation and Visualization. Blast+ and FastQC are installed as binaries and the other packages are compiled from source. The usage for the script is:

```
./build_3rd_party.sh -h
usage: build_3rd_party.sh [PREFIX] [CC] [CXX] [OPT]
```

NOTE We use this script internally to install and test BioLite, and we have only tested it on our own systems. It is likely that you will need to manually install additional dependencies on OS X (or use a system like macports, homebrew, or Fink), or install additional packages on Linux through your distro's package manager (especially some development packages that end in -dev or -devel).

To install to '/usr/local/', you can call the script with no arguments. To use a different install path, specify a PREFIX, for instance your home directory:

```
./build_3rd_party.sh $HOME
```

If you want to specify a different compiler, use the CC and CXX options. For instance, your linux distro may have a gcc 4.6 package that installs 'gcc46', so you would use:

```
./build_3rd_party.sh /usr/local gcc46 g++46
```

Finally, if you want to specify more aggressive compiler optimizations, use the OPT option. If you have a newer CPU that supports SSE4.2 instructions (e.g. Intel Nehalem), you could use:

```
./build_3rd_party.sh /usr/local gcc g++ -msse4.2
```

1.1.7 Generating a tarball from the git repo

The included release.sh script will update the git version, rebuild the documentation and run the necessary configure/make commands to create a tar ball.

To build the documentation, you must install the 'sphinx' Python package, for instance with:

```
sudo pip install sphinx
```

or:

```
sudo easy_install sphinx
```

1.2 Configuration

After successfully installing BioLite with *make install*, you should see a message like:

```
|-------------------------------------------------------------------------
| BioLite has been installed to /usr/local
|
| Your default configuration file is located at:
|
|   /usr/local/share/biolite/biolite.cfg
|
|-------------------------------------------------------------------------
```

pointing to the location of your default BioLite configuration file. This file serves as the default configuration for any user on the system. To override it on a per-user basis, simply copy the file to *$HOME/.biolite/biolite.cfg* and make any required changes.

You can also override the location of the configuration file with an environment variabl. In bash:

```
export BIOLITE_CONFIG=/your/path/to/biolite.cfg
```

or in csh:

```
setenv BIOLITE_CONFIG /your/path/to/biolite.cfg
```

Finally, the BIOLITE_RESOURCE environment variable allows you to temporarily override specific values in the resources section of the configuration. For instance, if your configuration file is set to 2 threads, but want to test out a run with 8 threads instead, you could use (in bash):

```
export BIOLITE_RESOURCES="threads=8"
```

The value of this variable can be a comma-separated list of key=value pairs.

1.3 Cataloging data

The easiest way to interact with the BioLite catalog is using the *catalog* script packaged wit BioLite:

```
$ catalog -h
usage: catalog [-h] {insert,all,search,sizes} ...

Command-line tool for interacting with the agalma catalog.

agalma maintains a 'catalog' stored in an SQLite database of metadata
associated with your raw Illumina data, including:

- A unique ID that you make up to reference this data set.
- Paths to the FASTQ files containing the raw forward and reverse reads.
- The species name and NCBI ID.
- The sequencing center where the data was collected.

optional arguments:
  -h, --help            show this help message and exit

commands:
  {insert,all,search,sizes}
    insert              Add a new record to the catalog, or overwrite the
                        existing record with the same id.
    all                 List all catalog entries.
    search              Search all fields (except 'paths') for entries
                        matching the provided pattern, which can include * as
                        a wildcard.
    sizes               List all paths in the catalog, ordered by size on
                        disk.
```

The documentation below describes the *catalog* module, for manually interacting with the catalog from within a Python script.

1.3.1 `catalog` Module

The BioLite *catalog* table pairs metadata with the raw NGS data files (identified by their absolute path on disk). It includes the following:

- A *unique ID* for referencing the data set. If the data is paired-end Illumina HiSeq data, the ID can be automatically generated using unique information in the Illumina header.
- *Paths* to the raw sequence data. For paired-end Illumina data, this is expected to be two FASTQ files (possibly compressed) containing the forward and reverse reads.
- *Notes* about the species, the sample preparation and origin, the species, IDs from NCBI and ITIS taxonomies, and the sequencing machine and center where the data were collected.

The catalog acts as a bridge between the BioLite diagnostics and a more detailed laboratory information management system (LIMS) for tracking provenance of sample preparation and data collection upstream of and during sequencing. It contains the minimal context needed to associate diagnostics reports of downstream analyses with the raw sequence data, but without replicating or reimplementing the full functionality of a LIMS.

class `biolite.catalog.`**`CatalogRecord`**

Bases: `tuple`

A named tuple for holding records from the *catalog Table*.

`extraction_id`
Alias for field number 5

`id`
Alias for field number 0

`itis_id`
Alias for field number 4

`library_id`
Alias for field number 6

`library_type`
Alias for field number 7

`ncbi_id`
Alias for field number 3

`note`
Alias for field number 11

`paths`
Alias for field number 1

`sample_prep`
Alias for field number 12

`seq_center`
Alias for field number 10

`sequencer`
Alias for field number 9

`species`
Alias for field number 2

`timestamp`
Alias for field number 13

tissue
Alias for field number 8

biolite.catalog.**split_paths**(*paths*)
Splits a catalog path entry to return a list of paths.

biolite.catalog.**insert**(***kwargs*)
Insert or update a catalog entry, where keyword arguments specify the column/value pairs. If an entry for the given ID already exists, then the specified column/values pairs are used to update the entry. If the ID does not exist, a new entry is created with the specified values.

biolite.catalog.**select**(*id*)
Returns a CatalogRecord object for the given catalog ID, or :keyword:None if the ID is not found in the catalog.

biolite.catalog.**select_all**()
Yields a list of CatalogRecord objects for all entries in the catalog, ordered with the default ordering that SQLite provides.

biolite.catalog.**search**(*string*)
Yields a list of CatalogRecord objects for all entries in the catalog with an indexed column matching the given search *string*. The indexed columns are all the columns in the catalog except *paths*.

biolite.catalog.**make_record**(***kwargs*)
Returns a CatalogRecord object by mapping the provided keyword arguments to field names.

1.4 Diagnostics

Diagnostics usually come in the form of plots or summary statistics. They can serve many purposes, such as:

- diagnosing problems in sample preparation and optimizing future preparations;
- providing feedback on the sequencing itself, e.g. on read quality;
- implementing 'sanity checks' at intermediate steps of analysis;
- finding optimal parameters by comparing previous runs;
- recording computational and storage demands, and predicting future demands.

The *diagnostics* database table archives summary statistics that can be accessed across multiple stages of a pipeline, from different pipelines, and in HTML reports.

A diagnostics record looks like:

```
catalog_id | run_id | entity | attribute | value | timestamp
```

The *entity* field acts as a namespace to prevent attribute collisions, since the same attribute name can arise multiple times within a pipeline run.

When running a BioLite pipeline, the default entity is the pipeline name plus the stage name, so that values can be traced to the pipeline and stage during which they were entered. Entries in the diagnostics table can include paths to derivative files, which can be summaries of intermediate files that are used to generate reports or intermediate data files that serve as input to other stages and pipelines.

1.4.1 Initializing

Before logging to diagnostics, your script must initialize this module with a BioLite catalog ID and a name for the run using the *init* method. This will return a new run ID from the *runs Table*. Optionally, you can pass an existing run ID to *init* to continue a previous run.

Diagnostics are automatically initialized by the Pipeline and IlluminaPipeline classes in the *pipeline Module*.

1.4.2 Logging a record

Use the *log* function described below.

Detailed system utilization statistics, including memory high-water marks and compute wall-time are recorded automatically (by the wrapper base class) for any wrapper that your pipeline calls, and for the overall pipeline itself.

1.4.3 Provenance

Because every wrapper call is automatically logged, the diagnostics table holds a complete non-executable history of the analysis, which complements the original scripts that were used to run the analysis. In combination, the diagnostics table and original scripts provide provenance for all analyses.

class `biolite.diagnostics.`**`OutputPattern`**
Bases: `tuple`

OutputPattern(re, entity, attr)

`attr`
Alias for field number 2

`entity`
Alias for field number 1

`re`
Alias for field number 0

`biolite.diagnostics.`**`timestamp`**`()`
Returns the current time in ISO 8601 format, e.g. `YYYY-MM-DDTHH:MM:SS[.mmmmmm][+HH:MM]`.

`biolite.diagnostics.`**`str2list`**`(`*data*`)`
Converts a diagnostics string with key *name* in *self.data* into a list, by parsing it as a typical Python list representation `[item1, item2, ... ]`.

`biolite.diagnostics.`**`get_run_id`**`()`
Returns the *run_id* (as a string)

`biolite.diagnostics.`**`get_entity`**`()`
Returns the current *entity* as a dot-delimited string.

`biolite.diagnostics.`**`init`**`(`*id*, *name*, *run_id=None*, *workdir='/Users/mhowison/code/biolite/doc'*`)`
By default, appends to a file *diagnostics.txt* in the current working directory, but you can override this with the *workdir* argument.

You must specify a catalog *id* and a *name* for the run. If no *run_id* is specified, an auto-incremented run ID will be allocated by inserting a new row into the *runs Table*.

Returns the *run_id* (as a string).

`biolite.diagnostics.`**`check_init`**`()`
Aborts if the biolite.diagnostics.init() has not been called yet.

`biolite.diagnostics.`**`merge`**`()`
Merges the diagnostics and program caches into the SQLite database.

`biolite.diagnostics.`**`load_cache`**`()`
Similar to a merge, but loads the local diagnostics file into an in-memory cache instead of the SQLite database.

Uses the filename specified with *name*, or the file *diagnostics.txt* in the current working directory (default).

biolite.diagnostics.**log**(*attribute*, *value*)
Log an *attribute*/*value* pair in the diagnostics using the currently set *entity*. The pair is written to the local diagnostics text file and also into the local in-memory cache.

biolite.diagnostics.**log_path**(*path*, *log_prefix=None*)
Logs a *path* by writing these attributes at the current *entity*, with an optional prefix for this entry: 1) the full *path* string 2) the full *path* string, converted to an absolute path by os.path.abspath() 3) the *size* of the file/directory at the path (according to *os.stat*) 4) the *access time* of the file/directory at the path (according to *os.stat*) 5) the *modify time* of the file/directory at the path (according to *os.stat*) 6) the *permissions* of the file/directory at the path (according to *os.stat*)

biolite.diagnostics.**log_dict**(*d*, *prefix=None*)
Log a dictionary *d* by calling *log* for each key/value pair.

biolite.diagnostics.**log_program_version**(*name*, *version*, *path*)
Enter the version string and a hash of the binary file at *path* into the programs table.

biolite.diagnostics.**log_program_output**(*filename*, *patterns=None*)
Read backwards through a program's output to find any [biolite] markers, then log their key=value pairs in the diagnostics.

A marker can specify an entity suffix with the form [biolite.suffix].

[biolite.profile] markers are handled specially, since mem= and vmem= entries need to be accumulated. These are inserted into a program's output on Linux systems by the preloaded memusage.so library.

You can optionally include a list of additional patterns, specified as OutputPattern tuples with:

(regular expression string, entity, attribute)

and the first line of program output matching the pattern will be logged to that entity and attribute name. The value will be the subexpressions matched by the regular expression, either a single value if there is one subexpression, or a string of the tuple if there are more.

biolite.diagnostics.**lookup**(*run_id*, *entity*)
Returns a dictionary of *attribute*/*value* pairs for the given *run_id* and *entity* in the SQLite database.

Returns an empty dictionary if no records are found.

biolite.diagnostics.**local_lookup**(*entity*)
Similar to *lookup*, but queries the in-memory cache instead of the SQLite database. This can provide lookups when the local diagnostics text file has not yet been merged into the SQLite database (for instance, after restarting a pipeline that never completed, and hence never reached a diagnostics merge).

Returns an empty dictionary if no records are found.

biolite.diagnostics.**lookup_like**(*run_id*, *entity*)
Similar to *lookup*, but allows for wildcards in the entity name (either the SQL '%' wildcard or the more standard UNIX '*' wildcard).

Returns a dictinoary of dictionaries keyed on [*entity*][*attribute*].

biolite.diagnostics.**lookup_by_id**(*id*, *entity*)

biolite.diagnostics.**lookup_entities**(*run_id*)

biolite.diagnostics.**lookup_run**(*run_id*)

biolite.diagnostics.**lookup_runs**(*id=None*, *name=None*, *order='ASC'*)

biolite.diagnostics.**lookup_prev_run**(*id*, *previous*)
If *previous* is an integer, tries to lookup the exit diagnostics of a previous run with that run ID. If *previous* is any string, To input the results from a previous pipeline run, use the (–previous, -p) argument with a 'RUN_SPEC',

which is either a specific run ID to lookup in the diagnostics, or the wildcard '*', meaning the latest of any previous run found in the diagnostics for the given catalog ID.

biolite.diagnostics.**dump**(*run_id*)

biolite.diagnostics.**dump_by_id**(*id*)

biolite.diagnostics.**dump_all**()

biolite.diagnostics.**list_runs**(*name=None*, *id=None*, *hidden=False*)

biolite.diagnostics.**hide_run**(*run_id*)

biolite.diagnostics.**unhide_run**(*run_id*)

biolite.diagnostics.**list_programs**()

biolite.diagnostics.**exit_profiler**(*start*)
Capture script resource usage, after a script run ends or as an exit handler if the script fails.

biolite.diagnostics.**register_exit_profiler**(*start*)

1.5 Building pipelines

1.5.1 `pipeline` Module

BioLite borrows from Ruffus (http://code.google.com/p/ruffus/) the idea of using Python function decorators to delineate pipeline stages. Pipelines are created with a sequence of ordinary Python functions decorated by a pipeline object, which registers each function as a *stage* in the pipeline. The pipeline object maintains a persistent, global dictionary, called the *state*, and runs each stage by looking up the argument names in the stage function's signature, and calling the function with the values in the state dictionary whose keys match the function's argument names. This is implemented using the function inspection methods available from the `inspect` module in the Python standard library. If the stage function returns a dictionary, it is *ingested* into the pipeline's state by adding values for any new keys and updating values for existing keys. Arguments passed on the command-line to the pipeline script form the initial data in the pipeline's state.

As an example, the following code setups a pipeline with two command-line arguments and one stage. Note how the variable names in the stage function's signature match the names of the arguments. The stage uses the *ingest* call to pull the *output* path into the pipeline's state. This way, it is accessible to other stages that might be added to this pipeline.

```
from biolite.pipeline import BasePipeline
from biolite.wrappers import FilterIllumina

pipe = BasePipeline('filter', "Example pipeline")

pipe.add_argument('input', short='i',
      help="Input FASTA or FASTQ file to filter.")

pipe.add_argument('quality', short='q', type=int, metavar='MIN',
      default=28, help="Filter out reads that have a mean quality < MIN.")

@pipe.stage
def filter(input, quality):
      '''
      Filter out low-quality and adapter-contaminated reads
      '''
      output = input + '.filtered'
      FilterIllumina([input], [output], quality=quality)
```

```
    ingest('output')

if __name__ == "__main__":
    pipe.parse_args()
    pipe.run()
```

This script is available in *examples/filter-pipeline.py* and produces the following help message:

```
$ python examples/filter-pipeline.py -h
usage: filter-pipeline.py [-h] [--restart [CHK]] [--stage N] [--input INPUT]
                          [--quality MIN]

Example pipeline

optional arguments:
  -h, --help            show this help message and exit
  --restart [CHK]       Restart the pipeline from the last available
                        checkpoint, or from the specified checkpoint file CHK.
  --stage N             Start at stage number N. Note that some stages require
                        the output of previous stages, so starting in the
                        middle of a pipeline may not work.
  --input INPUT, -i INPUT
                        Input FASTA or FASTQ file to filter.
  --quality MIN, -q MIN
                        Filter out reads that have a mean quality < MIN. [28]

pipeline stages:
  0) [filter]
     Filter out low-quality and adapter-contaminated reads
```

The pipeline module allows you to rapidly create full-featured pipeline scripts with help messages, checkpointing and restart capabilities, and integration with the BioLite diagnostics and catalog databases (using the *Pipeline* or *IlluminaPipeline* derived classes).

Meta-Pipelines

Modularity is a key design goal, and it is possible to reuse one or more stages of an existing pipeline when building a new pipeline. It is also possible to build meta-pipelines that connect together several sub-pipelines.

Checkpoints

The pipeline object also incorporates fault tolerance. At the end of each stage, the pipeline stores a *checkpoint* by dumping its current state to a binary file with the `cPickle` module. This way, if a run is interrupted, either due to an internal error or to external conditions, such as a kill signal from a batch system or a hardware failure, the run can be restarted from the last completed stage (or, optionally, from any previous stage in the checkpoint).

class `biolite.pipeline.`**`BasePipeline`**(*name*, *desc=''*)

BasePipeline is the more generic class. It is designed to be used independently of the BioLite diagnostics and catalog features.

`import_stages`(*pipe*, *start=0*)

`import_arguments`(*pipe*, *names=None*)

`import_module`(*module*, *names=None*, *start=0*)

Imports another pipeline module. Adds the pipeline as a subpipeline and links to the module itself so that it can be referenced later.

import_pipeline(*pipe*, *names=None*, *start=0*)
Imports another pipeline. This should only be used in cases where the pipeline is in the same file as another pipeline.

make_state(**args*)

get(*key*)

stage(*func*)
Decorator to add functions as stages of this pipeline.

add_stage(*func*)

list_stages()

size()
Returns the size of the pipeline (the number of stages it contains).

parse_args()
Reads values passed as arguments into the pipeline's *state*.

add_argument(*name*, ***kwargs*)
Adds an argument *–name* to the pipeline. The single character keyword argument 'short' is used as the short versino of the argument (e.g. `short='n'` for *-n*). All other keyword arguments are passed through to the ArgumentParser when *parse_args* is called.

checkpoint()
Writes checkpoint file by making a deep copy of the pipeline's current *state* and pickling it to the value of *chkfile* in the state (by default, this is the pipeline's name followed by '.chk' in the current working directory).

restart(*chkfile*)
Restart the pipeline from the last stage written to the checkpoint file *chkfile*, which is unpickled and loaded as the current *state* using a deepcopy.

run()
Starts the pipeline at the stage specified with *–stage*, or at stage 0 if no stage was specified.

rerun(*state*, *start=0*, *stdout=None*)
Starts the pipeline without loading the command line arguments (e.g. for calling a full pipeline from within the stage of another pipeline), and instead using the provided *state*.

The pipeline's stdout stream can be temporarily redirected to a log file using *stdout*.

run_stage(*func*)
Runs the current stage (from *self.nstage*) by using the `inspect` module to read the function signature of the decorated stage function, then injecting values from the *state* where the key matches the variable name in the function signature.

ingest(**args*)
Called from inside a pipeline stage to ingest values back into the pipeline's *state*. It uses the `inspect` module to get the calling functions (i.e. the stage function's) local variable dictionary, and copies the variable names specified in the *args* list.

class `biolite.pipeline.`**Pipeline**(*name*, *desc=''*)
Bases: `biolite.pipeline.BasePipeline`

Extends BasePipeline to make use of the BioLite diagnostics and catalog databases.

set_outdir()
Setup the output directory.

get_file()
Returns the absolute path to the file that this pipeline was created in.

get_all_files()
Returns a flat list of all the files this pipeline and subpipelines are created in.

run()

finish(**args*)

add_stage(*func*)

class biolite.pipeline.**IlluminaPipeline**(*name*, *desc=''*)
Bases: biolite.pipeline.Pipeline

An extension of Pipeline that assumes that the input model is a forward and reverse FASTQ pair, such as a paired-end Illumina data set.

import_stages(*pipe*, *start=1*)

1.6 Generating reports

1.6.1 report Module

Provides a framework for generating HTML reports from BioLite diagnostics. The typical usage is to extend the *BaseReport* class for each pipeline, and override the *init* method to specify **lookups** and **generators**.

Lookups are called with *self.lookup* and specify entities or attributes that should be loaded from the diagnotics into the *self.data* AttributeDict. For example:

```
self.lookup('args', diagnostics.INIT)
```

will load the initialization entity, which includes all of the command-line arguments passed to the pipeline for a given run.

Generators are functions that return lists of HTML lines, which are concatenated together to form the final HTML report, in the order that the generators are attached. A generator function will typically start by checking if a diagnostics value was successfully loaded into *self.data*, e.g.:

```
def report_arguments(self):
      if 'args' in self.data:
              html = [self.header('Arguments')]
              html += ['<p>%s</[>' % a for a in self.data.args]
              return html
```

The generator is attached to the report in the *init* method with the line:

```
self.generator(self.report_arguments)
```

class biolite.report.**Field**
Bases: tuple

Field(key, title, type, format)

format
Alias for field number 3

key
Alias for field number 0

title
Alias for field number 1

type
Alias for field number 2

biolite.report.**profile_aggregate**(*profiles*)
Applies aggregators (sum or max) to fields in the input profiles list.

Returns a dict of aggregated values.

class biolite.report.**BaseReport**(*id*, *run_id*, *outdir=None*, *verbose=False*, *hlevel=1*)
A base class that provides basic infrastructure for reporting diagnostics via HTML for a given run.

This is intended to be sub-classed within an BioLite pipeline script, to define how the diagnostics for that pipeline should be summarized and plotted.

init()
Override this function with a series of lookup() and generator() calls that specify the diagnostics lookups needed by your report, and the sub-class functions that generate the HTML output.

lookup(*name*, *entity*, *attribute=None*, *func=<function lookup at 0x105b899b0>*)
Lookup data from the diagnotics table for the given entity and store it in the self.data dictoinary.

query(*name*, *sql*, *args*, *database=<module 'biolite.database' from '/Users/mhowison/code/biolite/doc/biolite/database.pyc'>*)

extract_arg(*entity*, *arg*)
Parse the 'command' attrbute of 'entity' to find the value for the argument 'arg'.

generator(*func*)
Add functions in your sub-class to the 'generators' list, and their list-of-strings output will be appended in order to the output of the object's __repr__ function.

check(**args*)
Check if multiple keys are in the report's data dictionary. Return true if all exists, otherwise false.

zip(**args*)
Zip together multiple items from the report's data dictionary.

header(*html*, *level=0*)

percent(*data_name*, *field_name*, *num*, *div*)

str2list(*name*)
Converts a diagnostics string with key *name* in *self.data* into a list, by parsing it as a typical Python list representation `[item1, item2, ... ]`.

summarize(*schema*, *name*, *attr=None*)
Returns a 2-column summary table of a pipeline's key statistics.

table(*rows*, *headers=None*, *style=None*)
Returns an HTML table with a row for each tuple in *rows*, and an option header row for the tuple *headers*. The *style* string indicates justification for a column as either l (left) or r (right). For example, 'lr' prints a table with the first column left-justified and the second column right-justified.

histogram(*imgname*, *data*, *bins=100*, *props={}*)
Plots a histogram in dict *data* with the given number of *bins* to the file *imgname*. The keys of the dict should correspond to bins with width 1, and the values to frequencies.

histogram_categorical(*imgname*, *data*, *props={}*)
Plots a histogram in dict *data* to the file *imgname*, using the keys as categories and values as frequencies.

histogram_overlay(*imgname*, *hists*, *labels=None*, *bins=100*, *props={}*)
Plots up to 3 histograms over each other. Histograms are plotted in the order of the hists list, so that the last histogram is the topmost. The histograms are plotted with alpha=0.5 and colors red, blue, green.

barplot(*imgname*, *data*, *props={}*)
Plots bars for a dict *data* to the file *imgname*, using the keys as categories and values as heights.

scatterplot(*imgname*, *plot*, *props={}*)
Plots the (X,Y) points given in *plot* to the file *imgname*.

plot should be a tuple of the form *(x, y, ...)* where x and y are list or nparray objects and any additional fields are parameters to the matplotlib *plot* function (such as color or label).

multiscatterplot(*imgname*, *plots*, *props={}*)
Plots multiple sets of (X,Y) points given in *plots* to the file *imgname*.

plots should be a list of tuples of the form *(x, y, color, label)* where x and y are list or nparray objects, color is a matplotlib color specification (for instance, 'r' for red) and label is a string.

lineplot(*imgname*, *data*, *props={}*)
Plots a single line for the values in *data* to the file *imgname*.

multilineplot(*imgname*, *plots*, *props={}*)
Plots multiple lines, one for each *(x, y, label)* tuple in the *plots* list, to the file *imgname*.

profile_table()
CPU and memory usage for each command called by this pipeline.

1.7 Calling external tools

1.7.1 wrappers Module

A series of wrappers for external calls to various bioinformatics tools.

class biolite.wrappers.**BaseWrapper**(*name*)
A base class that handles generic wrapper functionality.

Wrappers for specific programs should inherit this class, call *self.init* to specify their *name* (which is a key into the executable entries in the BioLite configuration file), and append their arguments to the *self.args* list.

By convention, a wrapper should call *self.run()* as the final line in its *__init__* function. This allows for clean syntax and use of the wrapper directly, without assigning it to a variable name, e.g.

wrappers.MyWrapper(arg1, arg2, ...)

When your wrapper runs, BaseWrapper will do the following:

- log the complete command line to diagnostics;
- optionally call the program with a version flag (invoked with *version*) to obtain a version string, then log this to the *programs Table* along with a hash of the binary executable file;
- append the command's stderr to a file called *name*.log in the CWD;
- also append the command's stdout to the same log file, unless you set *self.stdout*, in which case stdout is redirected to a file of that name;
- on Linux, add a memory profiling library to the LD_PRELOAD environment variable;

- call the command and check its return code (which should be 0 on success, unless you specify a different code with *self.return_ok*), optionally using the CWD specified in *self.cwd* or the environment specified in *self.env*.
- parse the stderr of the command to find [biolite.profile] markers and use the rusage values from *utils.safe_call* to populate a profile entity in the diagnostics with walltime, usertime, systime, mem, and vmem attributes.

init(*name*)
A shortcut for calling the BaseWrapper __init__ from a subclass.

check_arg(*flag*, *value*)
If *value* evaluates to True, append *flag* and *value* to the argument list.

add_threading(*flag*)
Indicates that this wrapper should use threading by appending an argument with the specified *flag* followed by the number of threads specified in the BioLite configuration file.

add_openmp()
Indicates that this wrapper should use OpenMP by setting the $OMP_NUM_THREADS environment variable equal to the number of threads specified in the BioLite configuration file.

version(*flag=None*, *cmd=None*, *path=None*)
Generates and logs a hash to distinguish this particular installation of the program (on a certain host, with a certain compiler, program version, etc.)

Specify the optional 'binary' argument if the wrapper name is not actually the program, e.g. if your program has a Perl wrapper script. Set 'binary' to the binary program that is likely to change between versions.

Specify the optional 'cmd' argument if the command to run for version information is different than what will be invoked by *run* (e.g. if the program has a perl wrapper script, but you want to version an underlying binary executable).

version_jar()
Special case of version() when the executable is a JAR file.

run(*cmd=None*)
Call this function at the end of your class's *__init__* function.

run_jar(*mem=None*)
Special case of run() when the executable is a JAR file.

biolite.wrappers.**estimate_insert_size**()
For tools that need insert sizes, use available estimates from the diagnostics database, or resort to the default values in the BioLite configuration file.

Returns an AttributeDict with the fields *mean*, *stddev* and *max*.

class biolite.wrappers.**CountLines**(**inputs*)
Bases: biolite.wrappers.BaseWrapper

usage: count_lines [-t THREADS] [INPUT ...]

Count the number of lines in the INPUT files using multiple threads to increase throughput.

class biolite.wrappers.**Coverage**(*sam*, *stats*)
Bases: biolite.wrappers.BaseWrapper

usage: coverage [-i SAM] [-o STATS]

Parses a SAM alignment file and writes a coverage table to STATS with columns for the reference name, the length of the referene, and the number of reads covering it in the alignment.

class biolite.wrappers.**Exclude**(*exclude_files*, *input_files*, *output_files*, *keep=False*)
Bases: biolite.wrappers.BaseWrapper

usage: exclude -x EXCLUDE_FILE [-k] [...] [-i INPUT ...] [-o OUTPUT ...]

Filters all the reads in the input files (FASTA or FASTQ is automatically detected) and excludes those with ids found in any of the EXCLUDE_FILEs.

If multiple input files are specified, these are treated as paired files. So if a sequence in one input is excluded, its pair is also excluded from the same position in all other input files.

If the -k flag is specified, invert the selection to keep instead of exclude.

class biolite.wrappers.**Fastq2Fasta**(*fastq_path*, *fasta_path=None*, *qual_path=None*, *suffix=None*)
Bases: biolite.wrappers.BaseWrapper

usage: fastq2fasta -i FASTQ [...] [-o FASTA ...] [-q QUAL ...] [-a] [-t OFFSET] [-s SUFFIX]

Converts each FASTQ input file to a FASTA file and quality score file with the names <basename>.fasta and <basename>.fasta.qual, where <basename> is the name of INPUT up to the last period (or with the names FASTA and QUAL if specified).

FASTA and QUAL are *appended* to (not truncated).

class biolite.wrappers.**Fasta2Fastq**(*fasta_path*, *qual_path*, *fastq_path=None*)
Bases: biolite.wrappers.BaseWrapper

usage: fasta2fastq -i FASTA [...] -q QUAL [...] [-o FASTQ] [-a] [-t OFFSET]

Merges each FASTA file and its corresponding QUAL file into a FASTQ file with the name <basename>.fastq, where <basename> in the FASTA name up to the last period (or with name FASTQ if specified).

FASTQ is *appended* to (not truncated).

class biolite.wrappers.**FilterIllumina**(*inputs*, *outputs*, *unpaired_output=None*, *offset=None*, *quality=None*, *nreads=None*, *adapters=True*, *bases=True*, *sep=None*)
Bases: biolite.wrappers.BaseWrapper

usage: filter_illumina [-i INPUT ...] [-o OUTPUT ...] [-u UNPAIRED-OUTPUT] [-t OFFSET] [-q QUALITY] [-n NREADS] [-a] [-b] [-s SEP]

Filters out low-quality and adapter-contaminated reads from Illumina data.

If multiple input files are specified, these are treated as paired files. So if a sequence in one input is filtered, its pair is also filtered from the same position in all other input files.

class biolite.wrappers.**Interleave**(*inputs*, *output*, *sep=None*)
Bases: biolite.wrappers.BaseWrapper

usage: interleave -i INPUT [...] [-o OUTPUT] [-s SEP]

Interleaves the records in the input files (FASTA or FASTQ is automatically detected) and writes them to OUTPUT, or to stdout if no OUTPUT is specified.

class biolite.wrappers.**Randomize**(*input*, *output*, *order_mode=None*, *order_file='order.txt'*)
Bases: biolite.wrappers.BaseWrapper

usage: randomize [-i INPUT] [-o OUTPUT] [-r READ-ORDER] [-w WRITE-ORDER]

Randomizes the order of sequences in each INPUT file and writes these to a corresponding OUTPUT file. By default, a new random write order is generated and saved to WRITE-ORDER, if specified. Alternatively, specifying a READ-ORDER file uses that order instead of a random one.

class biolite.wrappers.**InsertStats**(*input*, *histogram=None*, *histogram_max=None*)
Bases: biolite.wrappers.BaseWrapper

usage: insert_stats -i SAM -o HIST -m MAX_INSERT

Reads a SAM alignment file and uses it to estimate the mean and std. dev. of the insert size of the mapped paired-end reads. A histogram of all insert sizes encountered is written to the HIST file.

class biolite.wrappers.**PileupStats**(*input*, *histogram=None*, *overlap=None*)
Bases: biolite.wrappers.BaseWrapper

usage: pileup_stats -i PILEUP -o HIST -m OVERLAP

Reads a SAMtools pileup file and uses it to find potential sequence disconnects. A histogram of all disconnect events encountered is written to the HIST file.

class biolite.wrappers.**FastQC**(*input*, *outdir*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for FastQC. http://www.bioinformatics.bbsrc.ac.uk/projects/fastqc/

class biolite.wrappers.**Dustmasker**(*input*, *output*, *window=None*, *level=None*, *linker=None*, *infmt='fasta'*, *outfmt='fasta'*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for dustmasker from NCBI Blast+. http://nebc.nerc.ac.uk/bioinformatics/docs/blast+.html

class biolite.wrappers.**Segmasker**(*input*, *output*, *window=None*, *locut=None*, *hicut=None*, *infmt='fasta'*, *outfmt='fasta'*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for segmasker from NCBI Blast+. http://nebc.nerc.ac.uk/bioinformatics/docs/blast+.html

class biolite.wrappers.**Blastn**(*query*, *db*, *out*, *outfmt=5*, *evalue=0.0001*, *targets=20*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for blastn from NCBI Blast. http://blast.ncbi.nlm.nih.gov/

class biolite.wrappers.**Blastp**(*query*, *db*, *out*, *outfmt=5*, *evalue=0.0001*, *targets=20*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for blastn from NCBI Blast. http://blast.ncbi.nlm.nih.gov/

class biolite.wrappers.**Blastx**(*query*, *db*, *out*, *outfmt=5*, *evalue=0.0001*, *targets=20*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for blastx from NCBI Blast. http://blast.ncbi.nlm.nih.gov/

class biolite.wrappers.**Rpsblast**(*query*, *db*, *out*, *outfmt=5*, *evalue=0.0001*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for blastn from NCBI Blast. http://blast.ncbi.nlm.nih.gov/

class biolite.wrappers.**MultiBlast**(*blast*, *threads*, *qlist*, *db*, *out*, *evalue=0.0001*, *targets=20*)
Bases: biolite.wrappers.BaseWrapper

usage: multiblast BLAST THREADS QUERY_LIST OUT [ARGS]

Runs a Blast PROGRAM (e.g. blastx, blastn, blastp) in parallel on a list of queries (in QUERY_LIST). Additional arguments to PROGRAM can be appended as ARGS.

The PROGRAM is called on each query with threading equal to THREADS. Recommendation: set THREADS to the number of cores divided by the number of query files.

The individual XML outputs for each query file are concatenated into a single output file OUT.

Example usage: multiblast blastn 4 “query1.fa query2.fa” all-queries.xml -e 1e-6

class biolite.wrappers.**MakeBlastDB**(*dbtype*, *in_name*, *db_name*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for makeblastdb from NCBI Blast. http://blast.ncbi.nlm.nih.gov/

class biolite.wrappers.**Bowtie2**(*inputs*, *mapping_file*, *output_path*, *local=True*, *sensitive=True*, *all=True*, *max_insert=None*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for the bowtie2 short-read aligner. http://bowtie-bio.sourceforge.net/

class biolite.wrappers.**Bowtie2Build**(*input_path*, *outdir_path*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for bowtie2-build component of Bowtie2. http://bowtie-bio.sourceforge.net/

class biolite.wrappers.**SamToBam**(*input_path*, *output_path*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**SamView**(*input_path*, *regions*, *output_path*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**SamSort**(*input_path*, *output_path*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**SamIndex**(*input_path*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**SamPileup**(*reference_path*, *bam_path*, *output_path*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**Trinity**(*inputs*, *outdir*, *max_insert=None*, *min_length=None*, *seq_type='fq'*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**ParallelButterfly**(*commands*)
Bases: biolite.wrappers.BaseWrapper

class biolite.wrappers.**Oases**(*outdir*, *ins_length=None*, *ins_length_sd=None*, *min_length=None*, *merge=False*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for Oases, a *de novo* transcriptome assembler. http://www.ebi.ac.uk/~zerbino/oases/

class biolite.wrappers.**VelvetH**(*inputs*, *outdir*, *kmer=61*, *merge=False*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for the velveth component of the Velvet *de novo* assember. http://www.ebi.ac.uk/~zerbino/velvet/

If *merge* is True, *input_path* must be a list of transcript FASTA files. Otherwise, *input_path* should a single FASTQ filename containing shuffled short reads or a list of FASTQ files where the first two form a paired file and the third is unpaired short reads.

class biolite.wrappers.**VelvetG**(*outdir*, *ins_length=None*, *ins_length_sd=None*, *min_length=None*, *merge=False*, *exp_cov='auto'*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for the velvetg component of the Velvet *de novo* assember. http://www.ebi.ac.uk/~zerbino/velvet/

class biolite.wrappers.**Macse**(*input*, *output*, *frameshift=-40*, *stopcodon=-150*)
Bases: biolite.wrappers.BaseWrapper

Multiple alignment of coding sequences.

class biolite.wrappers.**ParallelMacse**(*inputs*, *outputs*, *frameshift=-40*, *stopcodon=-150*, *workfile='paramacse.commands.txt'*)
Bases: biolite.wrappers.BaseWrapper

Multiple alignment of coding sequences, run in parallel.

class biolite.wrappers.**Raxml**(*input*, *output*, *model*, *output_dir*, *pars_rseed=None*, *extra_flags=None*)
Bases: biolite.wrappers.BaseWrapper

Maximum Likelihood based inference of phylogenetic trees.

class biolite.wrappers.**Gblocks**(*input*, *t='p'*, *b1=None*, *b2=None*, *b3=10*, *b4=5*, *b5='a'*)
Bases: biolite.wrappers.BaseWrapper

Selection of conserved block from multiple sequence alignments for phylogenetics.

class biolite.wrappers.**Mcl**(*input*, *output*, *inflation=2.1*)
Bases: biolite.wrappers.BaseWrapper

Analysis of networks.

class biolite.wrappers.**Phyutility**(*input*, *output*)
Bases: biolite.wrappers.BaseWrapper

A wrapper for Phyutilit: Analyses and modification on phylogenetic trees and phylogenetic matrices.

class biolite.wrappers.**TreePruning**(*input*)
Bases: biolite.wrappers.BaseWrapper

Tree manipulation to generate maximally inclusive subtrees with no more than one sequence per taxon.

1.8 Automating workflows

1.8.1 workflows Module

Provides a collection of helper functions that coordinate multiple wrappers from the *wrappers Module* to accomplish a unified goal or automate a common analysis task.

Workflows are available for the following groups of tasks:

- Assembly statistics and sweeps
- Contig parsing
- Blast result parsing
- SamTools automation
- Transcript cleaning

class biolite.workflows.**BlastHit**
Bases: tuple

BlastHit(query, title, definition, id, evalue, rank, orient, mask, score, bitscore, length, percent)

bitscore
Alias for field number 9

definition
Alias for field number 2

evalue
Alias for field number 4

id
Alias for field number 3

length
Alias for field number 10

mask
Alias for field number 7

orient
Alias for field number 6

percent
Alias for field number 11

query
Alias for field number 0

rank
Alias for field number 5

score
Alias for field number 8

title
Alias for field number 1

class biolite.workflows.**ContigHeader**
Bases: tuple

ContigHeader(locus, transcript, confidence, length)

confidence
Alias for field number 2

length
Alias for field number 3

locus
Alias for field number 0

transcript
Alias for field number 1

biolite.workflows.**blast_annotate_seqs**(*hits*, *fasta_in*, *hits_out*, *misses_out*, *all_out=False*, *rpkms={}*)
Iterates through the records in *fasta_in* and looks for a hit in a dict of BlastHit object, *hits*.

For each record with a hit, the RPKM (if provided), hit title, and evalue are added to the ID and the record is written to *hits_out*.

If there is no hit, the record is written to *misses_out*.

If *all_out* is True, then hits are also written to *misses_out*.

biolite.workflows.**blast_hits**(*xml_path*, *nlimit=None*)
Reads an XML formatted BLAST report, and yields one named tuple per alignment, i.e. per hit between a query and a subject. Each named tuple has the following elements:

query title definition id evalue rank orient mask score bitscore length percent

where:

- orient is 1 if query and subject are in the same direction, 2 if they are in the opposite direction, and 0 if direction is inconsistent across hsp's
- evalue is the minimum evalue across hsp's
- score, bitcore and length are maximal across hsp's

biolite.workflows.**blast_top_hits**(*xml_path*)
Similar to *blast_hits*, but returns an OrderedDict keyed by query name with only one hit (the top hit) per query.

biolite.workflows.**calculate_rpkms**(*coverage_table*)

biolite.workflows.**clean_rrna**(*fasta_in*, *clean_out*, *rrna_out*)
Blastn against rRNA, transferring sequences with or without a hit to their own files. Even when rRNA reads are removed prior to assembly, some may make it through and be assembled from the full dataset (including low frequency contaminant rRNAs).

biolite.workflows.**clean_swissprot**(*fasta_in*, *clean_out*, *annotated_out*, *blast_out*, *rpkms=None*)
Blastn against SwissProt, transferring sequences with or without a hit to their own files, used in comparing assemblies.

biolite.workflows.**clean_univec**(*fasta_in*, *clean_out*, *vector_out*)
Blastn against univec, transferring sequences with or without a hit to their own files This removes sequences that still have adapters, or that are contaminated with plasmids (including the protein expression plasmids used to manufacture sample prep enzymes).

biolite.workflows.**contig_stats**(*fasta_path*, *hist_path*)
Parses the assembled contigs in *fasta_path* and writes a histogram of contig length to *hist_path*.

Writes the total contig count, mean length, and N50 length to the diagnostics.

biolite.workflows.**dustmasker**(*fasta_in*, *clean_out*, *dirty_out*, *max_lowc=0.8*, *min_region=0.1*, *unpack_func=<function unpack_oases_header at 0x1069297d0>*)

biolite.workflows.**extract_oases_exemplars**(*input_path*, *output_path*, *min_length=0*)
Extracts a single exemplar transcript for each locus in an Oases assembly at *input_path* and writes it to *output_path*. Only transcripts longer than *min_length* are considered.

The exemplar is chosen as the transcript with the highest confidence score.

biolite.workflows.**filter_coverage_table**(*coverage_table*, *seq_ids*, *filtered_table*)
Filters a *coverage_table* so that only entries with IDs in the list *seq_ids* remain and writes output to the path *filtered_table*.

biolite.workflows.**multiblast**(*blast*, *query*, *db*, *out*, *evalue=0.0001*, *cores=4*, *targets=20*)
Prepares a single *query* file for the *multiblast* by dividing the queries into `nodes = threads/cores` many chunks, where *threads* is from the BioLite configuration file.

Executes the Blast operation *blast* (e.g. 'blastx') in parallel on each *node*, then concatenates the XML output into a single XML file *out*.

biolite.workflows.**oases_assemblies**(*inputs, kmers=[61], workdir='./', min_length=None, ins_length=None*)
Automates Oases assemblies that sweep multiple *kmers*.

If *inputs* is a list of FASTQ files, they are automatically shuffled together. Or, provide a singleton list with the path to a pre-shuffled FASTQ file.

biolite.workflows.**oases_clean**(*workdir='./'*)
Cleans up a work directory that was used for an Oases assembly.

biolite.workflows.**oases_concat_assembly**(*inputs*, *concat_path*, *kmers*, *workdir='./'*, *ins_length=None*)

Performs Oases assemblies sweeping over the provided *kmers* list, and concatenates all contigs to *concat_path*.

If *inputs* is a list of FASTQ files, they are automatically shuffled together. Or, provide a singleton list with the path to a pre-shuffled FASTQ file.

biolite.workflows.**oases_merge_assembly**(*inputs*, *merge_path*, *merge_kmer*, *kmers*, *min_length=None*, *workdir='./'*, *ins_length=None*)

Implements the Oases-M protocol for merging several Oases assemblies, as described in:

Schulz, M. H., Zerbino, D. R., Vingron, M., & Birney, E. (2012). Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels. Bioinformatics (Oxford, England), 1-7. doi:10.1093/bioinformatics/bts094

Performs Oases assemblies sweeping over the provided *kmers* list, then performs a Oases merge assembly with *merge_kmer*.

class biolite.workflows.**rRNAhit**

Bases: tuple

rRNAhit(locus, gene, confidence, orient, query)

confidence
Alias for field number 2

gene
Alias for field number 1

locus
Alias for field number 0

orient
Alias for field number 3

query
Alias for field number 4

biolite.workflows.**rrna_blast_hits**(*xml_path*, *unpack_header_func*)

Reads an XML formatted BLAST report, and saves one top hit per locus, using the transcript with the highest confidence for the locus.

The locus name and confidence are extracted from the query name with the supplied 'unpack_header_func' function.

Returns both a set of all the queries in the XML report, and a dictionary keyed by locus and storing the rRNA hits:

set(queries), dict(hits)

The rRNA hits are tuple with the following fields: (locus gene confidence orient query)

biolite.workflows.**sort_and_index_sam**(*sam_path*)

Uses SamTools to convert a SAM file at *sam_path* to BAM, then sort and index the BAM.

Returns the filename of the final output, which is '_sorted.bam' appended to *sam_path*.

biolite.workflows.**trinity_assembly**(*out*, *inputs*, *workdir='./'*, *min_length=None*)

biolite.workflows.**unpack_oases_header**(*header*)

Unpacks an Oases contig header into a ContigHeader object.

Example header:

>Locus_9919_Transcript_1/1_Confidence_1.000_Length_160

1.9 Internals

1.9.1 `config` Module

Loads the entries in the *biolite.cfg* file into two member dictionaries, *resources* (default parameters and data paths) and *executables* (paths to external executables called by *wrappers Module*).

By default, BioLite will look for the configuration file at the following paths, in order of preference:

- the value of the $BIOLITE_CONFIG environment variable
- $HOME/.biolite/biolite.cfg
- $PWD/biolite.cfg

`biolite.config.`**`init`**(*executables*, *resources*)
Called at module load to parse the BioLite configuration file.

`biolite.config.`**`get_resource`**(*key*)
Lookup a resource from the configuration file for *key* and print an intelligble error message on KeyError.

`biolite.config.`**`get_command`**(*key*)
Lookup the full path to an executable *key* in the configuration file and print an intelligble error message if the path can't be found in the user's PATH environment variable (similar to the Unix utility *which*).

The output is a list, starting with the full path to the executable, ready for input to subprocess.Popen. Any trailing parameters in config entry for the executable are preserved in this list.

If there is no config entry for the key, or the entry is blank, the key is used as the name of the executable. Thus, the config file only needs to override executable paths that won't resolve correctly using PATH.

1.9.2 `database` Module

Provides an interface to the underlying SQLite database that stores the BioLite **catalog**, **runs**, **diagnostics**, and **programs** tables.

catalog Table

```
CREATE TABLE catalog (
  id VARCHAR(256) PRIMARY KEY NOT NULL,
  paths TEXT,
  species VARCHAR(256),
  ncbi_id INTEGER,
  itis_id INTEGER,
  extraction_id VARCHAR(256),
  library_id VARCHAR(256),
  library_type VARCHAR(256),
  tissue VARCHAR(256),
  sequencer VARCHAR(256),
  seq_center VARCHAR(256),
  note TEXT,
  sample_prep TEXT,
  timestamp DATETIME);
CREATE INDEX catalog_species ON catalog(species);
```

```
CREATE INDEX catalog_ncbi_id ON catalog(ncbi_id);
CREATE INDEX catalog_itis_id ON catalog(itis_id);
CREATE INDEX catalog_extraction_id ON catalog(extraction_id);
CREATE INDEX catalog_library_id ON catalog(library_id);
CREATE INDEX catalog_library_type ON catalog(library_type);
CREATE INDEX catalog_tissue ON catalog(tissue);
CREATE INDEX catalog_sequencer ON catalog(sequencer);
CREATE INDEX catalog_seq_center ON catalog(seq_center);
CREATE INDEX catalog_note ON catalog(note);
CREATE INDEX catalog_sample_prep ON catalog(sample_prep);
CREATE INDEX catalog_timestamp ON catalog(timestamp);
```

runs Table

```
CREATE TABLE runs (
  run_id INTEGER PRIMARY KEY AUTOINCREMENT,
  id VARCHAR(256),
  name VARCHAR(32),
  hostname VARCHAR(32),
  username VARCHAR(32),
  timestamp DATETIME,
  hidden INTEGER DEFAULT '0');
CREATE INDEX runs_id ON runs(id);
CREATE INDEX runs_name ON runs(name);
```

diagnostics Table

```
CREATE TABLE diagnostics (
  id VARCHAR(256),
  run_id INTEGER,
  entity VARCHAR(128),
  attribute VARCHAR(32),
  value TEXT,
  timestamp DATETIME);
CREATE INDEX diagnostics_id ON diagnostics(id);
CREATE INDEX diagnostics_run_id ON diagnostics(run_id);
CREATE INDEX diagnostics_entity ON diagnostics(entity);
CREATE INDEX diagnostics_attribute ON diagnostics(attribute);
CREATE INDEX diagnostics_timestamp ON diagnostics(timestamp);CREATE UNIQUE INDEX entry ON diagnostics
```

programs Table

```
CREATE TABLE programs (
  binary CHAR(32) PRIMARY KEY NOT NULL,
  name VARCHAR(256),
  version TEXT);
CREATE INDEX programs_name ON programs(name);
```

biolite.database.**connect**(**args*, ***kwargs*)
Establish a gobal database connection and set the *execute* function.

biolite.database.**disconnect**()
Close the global database connection, set it to None, and set the *execute* function to auto-reconnect.

biolite.database.**execute**(**args*, ***kwargs*)
Establish a gobal database connection and set the *execute* function.

1.9.3 `utils` Module

Utility functions used by other BioLite modules.

biolite.utils.**die**(*message*, *retval=1*)
Prints the current BioLite module and an error *message*, then aborts.

biolite.utils.**info**(*message*)
Prints the current BioLite module and a *message*.

biolite.utils.**safe_mkdir**(*path*)
Creates the directory, including any missing parent directories, at the specified *path*.

Aborts if the path points to an existing regular file.

Returns the absolute path of the directory.

biolite.utils.**safe_remove**(*path*)
Removes a file at the given *path* only if it exists.

biolite.utils.**truncate_file**(*path*)
Truncates a file (i.e. overwrites with 0 bytes) at the given *path*.

biolite.utils.**rusage_diff**(*r1*, *r2*)
Returns an rusage object where each field is the difference of the corresponding fields in *r1* and *r2*.

biolite.utils.**failed_executable**(*executable*, *e*)
Diagnose why a wrapped executable failed to execute, and print an intelligble error message for the user.

biolite.utils.**safe_call**(**args*, ***kwargs*)
Calls an executable as a subprocess and checks the return value.

All *args* and *kwargs* are passed to a *subprocess.Popen* call, except for the special keywords *return_ok*, whose value is used to check the return value of the subprocess. By default, this is zero and any non-zero return is considered an error. To disable this check, set *return_ok* to *None*.

Returns a 3-tuple with the return code, the elapsed walltime, and an rusage structure with the elapsed usertime and systime.

biolite.utils.**safe_str**(*s*)
Returns the string *s* with only alpha-numerical characters and the special characters `()[]{}|:.-_` preserved. All other characters are replaced by _.

biolite.utils.**timestamp**()
Returns the current time in `YYYY-MM-DD HH:MM:SS` format.

biolite.utils.**safe_print**(*f*, *line*)
Places an exclusive lock around the file object *f* and writes *line* to it as an atomic write operation.

A line return is appended after *line*.

biolite.utils.**readlines_reverse**(*f*)
Seeks to the end of the file object *f* and yields lines in reverse order.

biolite.utils.**cat_to_file**(*input_path*, *output_path*, *mode='a'*, *start=0*)
Uses the **cat** or **awk** command to copy the contents at *input_path* to *output_path*, starting at line 0 of *input_path* and appending to *output_path* by default.

biolite.utils.**head**(*path*, *n=1*)
Returns a string with the first *n* lines of *path*.

biolite.utils.**head_to_file**(*input_path*, *output_path*, *n=1*, *mode='w'*)
Uses the **head** to copy the first *n* lines of *input_path* to *output_path*, overwriting the contents of *output_path* by default.

biolite.utils.**tail**(*path*, *n=1*)
Returns a string with the last *n* lines of *path*.

biolite.utils.**tail_to_file**(*input_path*, *output_path*, *n=1*, *mode='w'*)
Uses the **head** to copy the last *n* lines of *input_path* to *output_path*, overwriting the contents of *output_path* by default.

biolite.utils.**get_caller_info**(*depth=2*)
Uses the inspect module to determine the name of the calling function and its module.

Returns a 2-tuple with the module name and the function name.

biolite.utils.**get_caller_locals**(*depth=2*)
Uses the inspect module to return a dictionary of the local variables in the caller's frame at the given *depth*. The default *depth* of 2 corresponds to the frame that calls this function.

class biolite.utils.**AttributeDict**(**args*, ***kwargs*)
Bases: dict

A mutable alternative to namedtuple that supports accessing values as attributes or with the dict [] operator.

biolite.utils.**sorted_alphanum**(*l*)
Sorts a list of strings *l* and returns a list with the elements in alpha-numerical order (i.e. strings starting with numbers are correctly ordered by numerical value).

biolite.utils.**memusage**()
Reads the current memory usage for this process from /proc/self/status and returns two integer values *mem* and *vmem* which correspond to the VmHWM (max physical memory) and VmPeak (max virtual memory) fields.

Note: only works on Linux.

biolite.utils.**which**(*executable*)
Returns the full path to *executable* by searching through all entries in the $PATH environment variable, and looking for an executable file with that name.

Returns *None* if the executable is not found.

biolite.utils.**zipdir**(*dirname*)
Recursively zips all files in *dirname* into a zip archive with the name *dirname.zip* in the current working directory.

biolite.utils.**number_range**(*numbers*)
Collapse a list of numbers into a list of range strings, following http://stackoverflow.com/questions/9470611/how-to-do-an-inverse-range-i-

biolite.utils.**bytes_to_gb**(*b*)
Returns a string representing the given number of bytes as GB.

biolite.utils.**mem_to_mb**(*mem*)
Convert a memory string, like 2G or 100mb, to an integer number of megabytes.

CHAPTER

TWO

CITING

BioLite is still under development, and is an experimental tool that should be used with care. Please cite:

Howison, M., Sinnott-Armstrong, N. A., & Dunn, C. W. (2012, to appear). BioLite, a lightweight bioinformatics framework with automated tracking of diagnostics and provenance. In Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP '12).

We have not yet published a paper describing Agalma or any of the novel methods we introduce with this project, but we will. This project builds directly on a variety of things we learned in completing analyses for the following paper (though earlier prototype tools were used to execute these analyses):

Smith, SA, NG Wilson, F Goetz, C Feehery, SCS Andrade, GW Rouse, G Giribet, CW Dunn (2011). Resolving the evolutionary relationships of molluscs with phylogenomic tools. Nature. doi:10.1038/nature10526

If you use Agalma or BioLite in a published study, please contact us for an up-to-date citation. Or, check the publications page at the Dunn lab (http://dunnlab.org).

Agalma and BioLite makes use of many other programs that do much of the heavy lifting of the analyses. Please be sure to credit these essential components as well. Check the biolite.cfg file for web links to these programs, where you can find more information on how to cite them.

CHAPTER

THREE

FUNDING

This software has been developed with support from the following US National Science Foundation grants:

PSCIC Full Proposal: The iPlant Collaborative: A Cyberinfrastructure-Centered Community for a New Plant Biology (Award Number 0735191)

Collaborative Research: Resolving old questions in Mollusc phylogenetics with new EST data and developing general phylogenomic tools (Award Number 0844596)

Infrastructure to Advance Life Sciences in the Ocean State (Award Number 1004057)

The Brown University Center for Computation and Visualization has been instrumental to the development of BioLite.

CHAPTER

FOUR

INDICES AND TABLES

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

PYTHON MODULE INDEX

b

INDEX

G

H

I

K

L

M

N

W

Z