# Contents

1 What is Sequana? 3

2 User guide and reference 7
   2.1 Installation ................................................. 7
   2.2 User Guide ............................................... 11
   2.3 Tutorial ................................................. 15
   2.4 Pipelines ................................................ 18
   2.5 Gallery .................................................. 25
   2.6 Case Examples ........................................... 32
   2.7 Developer guide ......................................... 36
   2.8 Applications (standalone) ............................... 41
   2.9 References ............................................... 44
   2.10 FAQS ...................................................... 108
   2.11 Changelog ............................................... 110
   2.12 Glossary ............................................... 117
   2.13 Indices and tables ..................................... 118

Python Module Index 119
Current version: 0.1.21, Apr 10, 2017

**Python version**  Python 3.5 although some modules are Python2.7 compatible

**Source**  See http://github.com/sequana/sequana.

**Issues**  Please fill a report on github

**How to cite**  Desvillechabrol D, Bouchier C, Cokelaer T and Kennedy S. Sequana: a set of flexible genomic pipelines for processing and reporting NGS analysis [v1; no peer reviewed]. F1000Research 2016, 5:1767 (poster) (doi: 10.7490/f1000research.1112656.1)

For the coverage tool (sequana_coverage): Dimitri Desvillechabrol, Christiane Bouchier, Sean Kennedy, Thomas Cokelaer http://biorxiv.org/content/early/2016/12/08/092478
What is Sequana?

Sequana is a versatile tool that provides

1. A Python library dedicated to NGS analysis (e.g., tools to visualise standard NGS formats).
2. A set of pipelines dedicated to NGS in the form of Snakefiles (Makefile-like with Python syntax based on snakemake framework).
3. Original tools to help in the creation of such pipelines including HTML reports.
4. Standalone applications:
   
   (a) sequana_coverage ease the extraction of genomic regions of interest and genome coverage information
   
   (b) Sequanix: GUI for snakemake workflows, a GUI for Snakemake workflows (hence Sequana pipelines as well)

Currently, the available pipelines cover quality control (e.g. adapters removal, phix removal, trimming of bad quality bases), variant calling, characterisation of the genome coverage, taxonomic classification, de-novo assembly. See the Pipelines section for more information.

Sequana can be used by developers to create new pipelines and by users in the form of applications ready for production. A GUI will help users to change parameters and run the pipelines easily.

To join the project, please let us know on github.

![](images/sphx_glr_plot_coverage_001.png)
Chapter 1. What is Sequana?
**Installation**

**Installation using Conda**

If you have already installed Sequana dependencies, this command should install the latest release posted on [http://pypi.python.org/pypi/sequana](http://pypi.python.org/pypi/sequana) website:

```
pip install sequana --upgrade
```

If you have not installed Sequana, be aware that it relies on many dependencies that need to be compiled (i.e., it is time consuming and requires proper C compiler). For example, we use Matplotlib, Pandas that requires compilation. Besides, many pipelines rely on third-party software such as BWA or samtools that are not Python libraries.

**Install conda executable**

In practice, we do use Anaconda. We recommend to install conda executable via the manual installer. You may have the choice between Python 2 and 3. We recommend to choose a Python version 3.

**channels**

When you want to install a new package, you have to use this type of syntax:

```
conda install ipython
```

where ipython is the package you wish to install. Note that by default, conda looks on the official Anaconda website (channel). However, there are many channels available. We will use the bioconda channel. To use it, type these commands (once for all):
conda config --add channels conda-forge
conda config --add channels defaults
conda config --add channels r
conda config --add channels bioconda

**Warning:** it is important to add them in this order**, as mentionned on bioconda webpage

(https://bioconda.github.io/).

### Create an environment

Once **conda** is installed, open a new shell. Although this is not required strictly speaking, we would recommend to create an environment dedicated to Sequana. This environment can later be removed without affecting your system or conda installation. A **conda** environment is nothing else than a directory and can be created as follows:

```bash
conda create --name sequana python=3.5
```

Then, since you may have several environments, you must activate the **sequana** environment itself:

```bash
source activate sequana
```

Here are compulsory packages that must be installed:

```bash
conda install numpy matplotlib pandas snakemake graphviz pygraphviz scipy
```

Then, depending on the pipelines or standalone applications you want to use, you will need to install other packages. Here is a list of dependencies that should be enough to run most of the current pipelines (commands are split on several lines but you can also install everything in one go):

```bash
conda install pysam snpeff biokit bioservices spades khmer pyVCF
conda install bwa bcftools samtools bedtools picard freebayes fastqc
conda install kraken krona pigz
```

**Note:** the **denovo_assembly** pipelines uses Quast tool, which we ported to python 3.5 and was pulled on Quast official github page. This is not yet in bioconda but one can it from the quast github (sept 2016). This is required for the de-novo pipeline. The de novo pipeline also requires GATK, to be installed manually by users (due to licensing restrictions)

**Note:** **Sequana** is not fully compatible with Python 2.7 since a dependency (Snakemake) is only available for Python 3.5. However, many core functionalities would work under Python 2.7

**Note:** For **GATK** (variant caller), please go to https://software.broadinstitute.org/gatk/download/auth?package=GATK and download the file GenomeAnalysisTK-3.7.tar.bz2 ; then type:

```bash
gatk-register GenomeAnalysisTK-3.7.tar.bz2
```
**Docker containers for Sequana**

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run the software.

In order to allow anyone to use Sequana without needs for complex installation, we provide Docker images <https://hub.docker.com/u/sequana>, which are synchronized on the master branch of the source code.

We assume that:

1. You have installed Docker on your system (see Docker otherwise).
2. You have an account on Hub Docker.

**Quick start**

With your hub.docker account, first login:

```
docker login
```

Then download (pull) a Sequana image (all library, pipelines and standalones) as follows (2Gb image in total):

```
docker pull sequana/sequana
```

Now, you should be ready to try it. To start an interactive session, type:

```
cd <Directory_with_data>
docker run -v $PWD:/home/sequana/data -it sequana/sequana
```

**Standalone**

The primary goal of the docker is to make it possible to quickly test the standalones. For now, we expose only one docker. Please see specific documentation following the links here below:

- sequana_coverage_
- sequana_taxonomy

**More advanced Usage**

Here below, we provide a quick tutorial that will guide you on using Sequana thanks to the docker. To do so, we will focus on one standalone application called sequana_coverage. In brief, the standalone takes as input a BED file that contains the genome coverage of a set of mapped DNA reads onto a reference genome. Then, the standalone creates a report with relevant information about the coverage (See Sequana documentation for more information).

**Use the sequana Docker image**

Once you downloaded the sequana image, you can then enter into the image as follows:

```
docker run -it sequana/sequana
```

This opens an interactive shell with latest sequana library pre-installed. For instance, you can start an IPython shell:

```
ipython
```
and import the library:

```python
import sequana
```

See sequana.readthedocs.org for examples.

Or within the unix shell, you can use standalones. For instance there is a test BED file that can be analysed as follows to get a coverage report:

```bash
sequana_coverage --input virus.bed
```

This should print information and create a report/ directory. This is not very practical if you have your own files or want to open the HTML page stored in ./report. So, let us quit the docker:

```bash
exit
```

and do it the proper way. Go to a working directory (or your computer) and start the docker image again as follows:

```bash
docker run -v $PWD:/home/sequana/data -it sequana/sequana
```

This should start the docker image again but you should now have a `./data` directory. **Be aware that if you modify data here (in the image), you will also modify the data in your local data file.**

Now, you can run sequana_coverage in this directory:

```bash
cd data
sequana_coverage --input yourfile.bed
```

This analyses the data and creates a report/ directory. The container has no display but you can now go back to your computer in /home/user/mydatapath and browse the HTML page that was created.

Each time, we entered in the image but you can also use the images as executables (see standalone section above).

**For developers:**

Build the image:

```bash
git clone https://github.com/sequana/sequana
cd sequana/docker/sequana_core
sudo docker build -t="sequana/sequana_core" .
```

Run the image:

```bash
sudo docker run -it sequana/sequana_core
```

**Layers**

Here are the layers made available on hub.docker.com/u/sequana organizations. Each layer is built on top of the previous one

- sequana_core (only ubuntu + some packages)
- sequana_conda_core (sequana_core + conda + common scientific packages)
- sequana_conda_ngs (sequana_conda_core + NGS conda packages)
- sequana (sequana_conda_ngs + sequana specific version)
• Standalone Layers:
  – **sequana_coverage** (sequana + sequana_coverage standalone)

**Sudo**

To avoid using sudo, check out various forum. See for example: [http://askubuntu.com/questions/477551/how-can-i-use-docker-without-sudo](http://askubuntu.com/questions/477551/how-can-i-use-docker-without-sudo)

**For developers**

For the documentation:

```
pip install sphinx sphinx_rtd_theme pytest pytest-qt pytest-mock
```

**User Guide**

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>User Guide</strong></td>
</tr>
<tr>
<td>– <strong>Sequana library</strong></td>
</tr>
<tr>
<td>• Example 1: running median on coverage</td>
</tr>
<tr>
<td>• Example 2: read a fastq file</td>
</tr>
<tr>
<td>– <strong>Sequana standalones</strong></td>
</tr>
<tr>
<td>– <strong>Sequana pipelines</strong></td>
</tr>
<tr>
<td>• Manually</td>
</tr>
<tr>
<td>• Using sequana standalone</td>
</tr>
<tr>
<td>• Using Sequanix standalone</td>
</tr>
<tr>
<td>– <strong>Sequana Reports</strong></td>
</tr>
</tbody>
</table>

**Sequana** provides standalone applications (e.g., **sequana_coverage**, **sequana_taxonomy**) and pipelines in the form of Snakefiles. Although the standalone applications are usually simpler, they may not have all features or parameters offered by the pipelines.

The Tutorial, Pipelines, Gallery and Case Examples sections provide many examples on their usage.

This section will not describe all available standalones and pipelines. We will focus on one example (coverage) to show how one can use the **Sequana** library, or standalone application, or pipeline to get information about the coverage of a set of mapped reads onto a reference.

**Sequana library**
Example 1: running median on coverage

**Sequana** is a Python library. It contains many functionalities, which are fully documented and available in the *References* section. We can first look at the coverage contained within a BED file using the library. First, we need some data. **Sequana** provides some test examples, which can be accessed using `sequana_data()` function. The test case is a virus (about 18,000 bases):

```python
from sequana import sequana_data
filename = sequana_data('JB409847.bed')
```

We can then use the *GenomeCov* class to read the file:

```python
from sequana import GenomeCov
gc = GenomeCov(filename)
```

Select a chromosome (first one) and compute the running median:

```python
chrom = gc[0]
chrom.running_median(n=5001, circular=True)
chrom.compute_zscore()
```

and finally plot the coverage together with confidence interval (3 sigma):

```python
chrom.plot_coverage()
```
Example2: read a fastq file

Let us use the `FastQC` class to get the distribution of the bases ACGT across all reads of a FastQ file.

![Distribution of bases ACGT across reads](image)

Many more functionalities are available. The reference guide should help you.

**Sequana standalones**

The Python example about the coverage is actually quite useful. We therefore decided to provide a standalone application. There are other standalone applications listed in the `Applications (standalone)` section.

The one related to the coverage example shown above is named `sequana_coverage`. If you have a BED file, type:

```
sequana_coverage -i <BEDFILENAME>
```

If your organism has a circular DNA, add `-o`. You can play with the window size for the running median using `-w`.

Using the BED file and reference mentioned in the previous section you should obtain the same figure as above.

An additional feature is the report using `--show-html` option.

**Sequana pipelines**

In `Sequana`, in addition to the library and standalone applications, we also provide a set of pipelines (see the `Pipelines` section). The coverage tools described so far do not have a dedicated pipeline but is part of a more general pipeline.
called **Variant Calling**. Instead of describing in details that pipeline, let us explain the way pipelines can be created and run.

### Manually

Pipelines are made of a Snakefile (a Makefile using Python) and an associated config file. Pipelines can be downloaded from the [Sequana pipeline directory](https://github.com/biocompute-sequana/sequana) as well as the config file named `config.yaml`.

Copy the pipeline (ending in `.rules`) and the configuration file in a local directory. The config file is a generic template file and some fields must be changed. For instance the beginning of the file looks like:

```yaml
# list of your input file
samples:
  file1: "%(file1)s"
  file2: "%(file2)s"
```

For pipelines that takes FastQ files as inputs, the string `%(file)s` must be replaced by a valid filename. If you do not have a second file, remove the next line (file2). Other similar fields must be filled if required by the pipeline.

Then, a pipeline must be executed using the executable `snakemake`. If you choose the **variant_calling** pipeline, the file is executed as follows:

```bash
snakemake -s variant_calling.rules
```

This will search for the `config.yaml` file locally. One good feature is that if you interrupt the pipeline (or if it fails), you can fix the problem and re-run the command above without executing the parts of the pipelines that were successfully run. If you want to start from scratch, add `--forceall` option:

```bash
snakemake -s variant_calling.rules --forceall
```

**See also:** 

*Pipelines* section for more information.

### Using sequana standalone

An easier way to initialise a pipeline, is to use `sequana` executable. For instance for the variant calling:

```bash
sequana --pipeline variant_calling
```

This will automatically download the pipeline, config file and update the latter as much as possible.

**See also:**

*Applications (standalone)* section

### Using Sequanix standalone

An even easier way is to use our graphical interface named [Sequanix](https://github.com/biocompute-sequana/sequanix). A snapshot can be found in the *Sequanix: GUI for snakemake workflows* section.
Sequana Reports

Pipelines and standalone make use of internal reporting. Since there are part of the Sequana library, they can also be used with your own code. For instance, if you have a BAM file, you can use the following code to create a basic report:

```python
from sequana import BAM, sequana_data, BAMReport
b = BAM(sequana_data("test.bam", "testing"))
r = BAMReport()
r.set_data(b)
r.create_report()
```

that results can be shown in report/bam.html

Tutorial

Following the introductory example in User guide and reference, we will look at other pipelines such as the taxonomic classification, variant calling and coverage.

The following example will show how to run the quality control on a pair of FastQ files. The data comes from a sequencing (using HiSeq technology) of a Measles virus. For testing purposes, you can download R1 and R2 files that contain only 1500 reads. Copy them in a local directory.

Quality pipeline

Sequana comes with standalone applications and pipelines in the form of Snakefile (snakemake)

The following example will show how to initialise and run the quality control pipeline on a pair of FastQ files. The data comes from a sequencing (using HiSeq technology) of a Measles virus. For testing purposes, you can download R1 and R2 files that contain only 1500 reads. Copy them in a local directory.

First, run the sequana standalone application to initialise the pipeline quality_control:

```bash
sequana --pipeline quality_control --output-directory TEST --adapters PCRFree
```

This command downloads the required configuration file(s) in particular the config file and the pipeline itself. This example should work out of the box but you may want to look at the configuration file config.yaml. For instance, you may want to change the reference to the phix (by default we use phix174.fa, which is provided in Sequana) or change the adapter_removal section to your needs (cutadapt parameters, in particular the forward and reverse complement list of adapters; None by default).

By default, the output directory is called analysis and can be overwritten with the --output-directory parameter. Then, run the pipeline and wait for completion:

```bash
cd TEST
snakemake -s quality_control.rules --stats stats.txt -p -j 4 --forceall
```

The -p option shows the commands, -j 4 means use 4 threads when possible. Alternatively, there is also a runme.sh script.

You should now have a directory with a HTML report corresponding to the sample:
Sequana, Release 0.1.21

open Hm2_GTGAAA_L005/report_qc_Hm2_GTGAAA_L005/summary.html

See User guide and reference

Taxonomy

Download a toy kraken database designed for this problem (contains only 100 FASTA files mixing measles viruses and others viruses):

```python
from sequana import KrakenDownload, sequana_config_path
db = KrakenDownload()
db.download("toydb")
database_path = sequana_config_path + "/kraken_toydb"
```

Then, you may use a Sequana pipeline (see pipeline_taxon and sequana.kraken) or this standalone application:

```bash
sequana_taxonomy --file1 Test_R1.cutadapt.fastq.gz
       --file2 Test_R2.cutadapt.fastq.gz --database <database_path>
```

where `<database_path>` must be replaced with the proper path. Open the local HTML file krona.html. An example is available in Krona example.

Variant calling

Note that this does the variant calling + snpEff + coverage. See more information in the Variant Calling section.

Initialise the pipeline

Call sequana standalone as follows:

```bash
sequana --pipeline variant_calling --input-directory . --output-directory TUTORIAL
```

Go to the project directory

```bash
cd TUTORIAL
```

Get the genbank reference

Assuming the reference is K01711.1 (Measles virus), we first need to fetch the genbank file from NCBI:

```python
from bioservices import EUtils
eu = EUtils()
data = eu.EFetch(db="nuccore", id="K01711.1", rettype="gbwithparts", retmode="text")
with open("measles.gbk", "w") as fout:
    fout.write(data.decode())
```

Get the FASTA reference

We will also get the FASTA from ENA:
from bioservices import ENA
en = ENA()
data = ena.get_data('K01711', 'fasta')
with open("measles.fa", "w") as fout:
    fout.write(data.decode())

New in v0.10

Assuming the genbank and reference have the same name, you can simply type:

from sequana.snpeff import download_fasta_and_genbank
download_fasta_and_genbank("K01711", "measles")

Get a snpEff config file and update it

Then you need to initialise a config file for snpEff tool:

from sequana import snpeff
v = snpeff.SnpEff("measles.gbk")

Update the snpEff config file

Edit the config file config.yaml and add the filename measles.gbk in the snpEff section:

# snpEff parameter
snpeff:
    do: yes
    reference: "measles.gbk"

and bwa_ref section:

# Bwa parameter for reference mapping
bwa_mem_ref:
    reference: "measles.fa"

Run the pipeline

snakemake -s variant_calling.rules --stats stats.txt -p -j 4 --forceall

De novo

The denovo_assembly pipeline can be initialised in the same way:

sequana --pipeline denovo_assembly --input-directory . --output-directory denovo_test

Go to the denovo_test directory and edit the config file.
Warning: this is very time and computationally expensive. The digital_normalisation section is one that controls the memory footprint. In particular, you can check change max-tablesize to a small value for test-purposes (set the value to 3e6)

Pipelines

In Sequana parlance, a pipeline is an application based on Snakemake that consists of a Snakefile and a configuration file. Each pipeline and its configuration file can be automatically downloaded using:

```
sequana --pipeline <name>
```

By default the previous command creates a directory named analysis where the pipeline and config file are stored. The pipeline must not be changed but the configuration file can be edited to change the options.

Although the configuration is documented and should be self content, additional help for users and developers can be found for each pipeline in the following links.

Todo

the following sections are in progress but should already give useful information about the pipelines that are available.

denovo_assembly

**Overview** Denovo Assembly from FastQ files

**Input** FastQ file(s) from Illumina Sequencing instrument

**Output** FastA, VCF and HTML files

**Usage**

Example:

```
sequana --pipeline denovo_assembly --file1 R1.fastq.gz --file2 R2.fastq.gz --project_denoovo
cd denovo
snakemake -s denovo_assembly.rules -p --stats stats.txt -j 4
```

**Requirements**

- khmer
- quast
- spades
- [variant_calling]
Details

The reads normalization is performed with khmer (digital normalization). It is an optional task defined in the config file. You might use normalization if the sequencing depth (depth of coverage) is too important. Then, the denovo assembly is done with spades. Finally, the coverage and misassembly are evaluated with the variant calling pipeline.

Rules and configuration details

Here is a documented configuration file `../sequana/pipelines/denovo_assembly/config.yaml` to be used with the pipeline. Each rule used in the pipeline may have a section in the configuration file. Here are the rules and their developer and user documentation.

`samtools_depth`

`bwa`

`digital_normalisation`

`format_contigs`

`freebayes`

`quast`

`snpeff`

`spades`

Quality control

Overview Quality control, trimming (adapter removal) and taxonomic overview

Input A set of FastQ (paired of single-end)

Output fastqc, cleanup FastQ files

Usage

```
sequana --pipeline quality_control --input-directory . --output-directory analysis
```

Requirements

Requirements

- bwa
- samtools
- kraken
Details

The adapters are removed using cutadapt. If one specifies the quality trimming option in the config file, then we trim low-quality ends from reads BEFORE adapter removal.

The quality trimming algorithm is the same as in BWA. That is: subtract the cutoff (e.g. 30) from all qualities; compute partial sums from the end of the sequence; cut the sequence at the index at which the sum is minimal.

```
# Original qualities
42, 40, 26, 27, 8, 7, 11, 4, 2, 3
# Subtracting the threshold gives:
32, 30, 16, 17, -2, -3, 1, -6, -8, -7
# Partial sum from the end. Stop early if the sum is greater than zero:
(70), (38), 8, -8, -25, -23, -20, -21, -15, -7
```

Minimum is -25, we keep the bases 1,2,3,4:

```
42, 40, 26, 27
```

Another important point is that all searches for adapter sequences are error tolerant (allowing errors such as mismatches, insertions and deletions). The level of error tolerance is 10% by default.

Rules and configuration details

Here is a documented configuration file `../sequana/pipelines/quality_control/config.yaml` to be used with the pipeline. Each rule used in the pipeline may have a section in the configuration file. Here are the rules and their developer and user documentation.

**FastQC**

**Cutadapt**

**Kraken**

**RNA-seq**

- **Overview**  RNASeq
- **Input**  FastQ raw data from Illumina Sequencer (either paired or not)
- **Output**
- **Config file requirements**
Usage

Example:

```bash
sequana --pipeline rnaseq --input-dir . --output-directory analysis --adapters TruSeq
cd analysis
srun snakemake -s rnaseq.rules --stats stats.txt -p -j 12 --nolock --cluster-config
   --cluster_config.json --cluster "sbatch --mem={cluster.ram} --cpus-per-task={threads}"
```

Requirements

- cutadapt
- picard-tools
- bowtie
- bowtie2
- multiqc
- STAR
- fastq_screen
- featureCounts [subread]

Details

Rules and configuration details

Here is a documented configuration file `../sequana/pipelines/rnaseq/config.yaml` to be used with the pipeline. Each rule used in the pipeline may have a section in the configuration file. Here are the rules and their developer and user documentation.

FastQC

Fastq_screen

Cutadapt

Mapping on rRNA

Mapping on reference genome

Bowtie1

STAR

Counting
Variant Calling

Overview  Variant calling

Input  fastq file from Illumina Sequencing instrument

Output  vcf and html files

Config file requirements

- samples:file1
- samples:file2
- project
- reference:reference.fasta

Details

Snakemake variant calling pipeline based on pipelines of Varun Khanna (https://github.com/khannavarun) and Adrien Villain (https://github.com/avillain). Reads (paired or single) are mapped using bwa mem. Aligned reads are processed with picard tools markduplicates and GATK indel realigner. Freebayes is used to detect SNPs and short INDELs. An annotation file can be set to annotate detected variants. Variants are reported in a HTML report.

The pipeline provides a coverage analysis of the mapping coverage after bam processing. Coverage for each base position is computed with bedtools genomecov. Sequana provides a HTML report with dynamics plots of sequencing coverage and shows interesting regions which have unusual coverage depth.

Usage

```bash
sequana --pipeline variant_calling --file1 R1.fastq.gz --file2 R2.fastq.gz --project variant
--variant --reference reference.fasta
```

```bash
snakemake -s variant_calling.rules -p --stats stats.txt -j 4
```

Requirements

- gatk (genomeAnalysisTK)
- picard (picard-tools)
- bedtools
- bwa
- freebayes
- snpEff
- samtools
Compressor

**Overview** compressor can be used to compress/uncompress FastQ files recursively. It was designed to uncompress gzipped files and to compress them back into a bzip2 format. It was then extended to dsrc and non-compressed files. Supported formats are gz and bz2 and dsrc (http://sun.aei.polsl.pl/dsrc/download.html).

**Input** Any number of FastQ files (compressed or not)

**Output** The input files (compressed or not)

**requirements** gzip and bzip and their parallel versions (pigz and pbzip2) as well as dsrc (DNA compression tool).

**Usage**

A standalone named `sequana_compressor` is provided. Please see:

```
sequana_compressor --help
```

to get detailed information about the arguments. The following example converts all file ending in fastq.gz into a new compression format (bz2). Note that “fastq” before the extension is required:

```
sequana_compressor --source fastq.gz --target fastq.bz2
```

If you want to add recursivity, add the **recursive** argument. On a distributed system (e.g. slurm), you should use the **snakemake-cluster**. For example on SLURM, add:

```
--snakemake-cluster "sbatch --qos normal"
```

`compressor` allows one to go from one format in (fastq, fastq.gz, fastq.bz2, fastq.dsrc) to any format in the same list. So this is a fully connected network as shown below:

```
2.4. Pipelines
```
Here is another example:

```bash
sequana_compressor --source fastq.gz --target fastq.bz2 --threads 8
--snakemake-cluster "sbatch --qos normal" --recursive --jobs 20
```

The number of jobs is set to 4 by default and limited to 20 to have a reasonable IO access. You can use more using the `--bypass` argument. If nodes have 8 CPUs, use `threads=8`, this means 20 nodes will be used.

**Warning:** During the conversion, a `.snakemake` is created in each processed directory. If you interrupt the process, `snakemake` locks the directory. If you get an error message about locked directories, relaunch your previous command with `--unlock` to unlock the directories and start again.

**Requirements**

Parallel version of gzip and bzip, as well as dsr:

- pigz
- bzip2
- dsr
- bunzip2

**Config file**

In principle you should not use any config file if you use the standalone. Note, however, the format of the underlying config file (for pbzip2 and pigz, the number of threads is automatically set to the number of available threads).
DAG

Rules used by the pipeline

Depending on the value of the target and source, only one rule is included in the pipeline. For example if your source is fastq.gz and the target fastq.bz2, the gz_to_bz2 rule is included. Its documentation is here below:

Others similar rules that convert from one compressed format to another compressed formats are:

Gallery

General-purpose examples for sequana library. The following examples illustrates how Sequana library itself can be used to read and create some plots used within the pipelines.

BAM module example

Plot histogram of MAPQ values contained in a BAM file

```python
from sequana import BAM, sequana_data
```

Get a data set (BAM file) for testing

```python
from sequana import BAM, sequana_data
datatest = sequana_data('test.bam', "testing")
```

Use sequana.bamtools.BAM class to plot the MAPQ histogram

```python
b = BAM(datatest)
b.plot_bar_mapq()
```
Quality histogram a la fastQC

Get a data set example

```python
from sequana import sequana_data
dataset = sequana_data("test.fastq", "testing")
```

Create a FastQC instance

```python
from sequana import FastQC
qc = FastQC(dataset, verbose=False)
```

plot the histogram

```python
qc.boxplot_quality()
```
Total running time of the script: (0 minutes 0.277 seconds)
Download Python source code: plot_fastqc_hist.py
Download IPython notebook: plot_fastqc_hist.ipynb

Coverage module example

```python
from sequana import GenomeCov
from sequana import sequana_data
bedfile = sequana_data("JB409847.bed")
```

Reading input BED file

```python
gc = GenomeCov(bedfile)
```

Select a chromosome (first and only one in that example)

```python
chrom = gc[0]
print(chrom)
```

Out:

```
Genome length: 19795
Sequencing depth (DOC): 931.31
```
Sequana, Release 0.1.21

Sequencing depth (median): 988.00
Breadth of coverage (BOC) (percent): 96.60
Genome coverage standard deviation: 237.15
Genome coverage coefficient variation: 0.25

Compute running median and zscore telling the algorithm that the chromosome is circular.

```python
chrom.running_median(n=5001, circular=True)
chrom.compute_zscore()
p = chrom.get_centralness()
```

Out:

```
0.868451629199
```

Plotting

```python
chrom.plot_coverage()
```

Total running time of the script: (0 minutes 1.234 seconds)

Download Python source code: plot_coverage.py
Download Jupyter notebook: plot_coverage.ipynb
Kraken module example

In **Sequana**, we provide tools to quickly assess the taxonomic content of a reads (FastQ). It is based on the Kraken and Krona software. **Sequana** bridges the gap between those tools, Kraken databases and a simple interface to get a quick taxonomic overview of the FastQ content.

For more information, please see the `sequana.kraken` module documentation. Note that this feature is also part of the Quality control pipeline.

Although we will use the **Sequana** library hereafter, note that there is also a standalone application named `sequana_taxonomy`.

**Context**

running the kraken analysis takes some time so we do provide an output directly. The output file can be analysed within Sequana to plot a Pie chart but also Javascript Krona results. The kraken format is as follows:

```
C HISEQ:426:C5T65ACXX:5:2301:5633:7203 11234 203 0:2 11234:1 0:1 11234:1 0:2
   →11234:1 0:13 11234:1 0:1 11234:1 0:3 11234:1 0:16 11234:1 0:5 11234:1 0:6 11234:1
   →0:13 A:31 0:33 11234:1 0:7
C HISEQ:426:C5T65ACXX:5:2301:5815:7120 11234 203 0:4 11234:1 0:12 11234:1 0:22
   →11234:1 0:1 0 11234:1 0:5 11234:1 0:7 11234:1 0:5 A:31 0:3 11234:1 0:22 11234:1
   →0:18 11234:1 0:24 11234:1
```

Each row correspond to a read in the FastQ file(s). The first column is either C (classified) or U (unclassified) and the third column contains the taxon the most relevant.

The taxon are not readable so we first need to get the scientific names. Besides, the lineage would be useful. This is done in Sequana using the `sequana.kraken.KrakenResults`. See following example.

**Example**

In the following example, we use the results of a kraken analysis. The original toy data files contains 1500 reads mostly related to Measles virus

```python
from sequana import KrakenResults, sequana_data
test_file = sequana_data("test_kraken.out", "testing")
k = KrakenResults(test_file, verbose=False)
df = k.plot(kind='pie')
print(df)
```
Measles virus strains and isolates

```
[11234, 351680, 884099, 70149, 1056492, 884098, 1089795, 884095, 884097, 170528] others 1.534356
Measles virus strain Edmonston-Zagreb 5.737158
Measles virus strain MVi/California.USA/8.04 6.337558
Measles virus genotypes and isolates 6.871247
Unclassified 12.274850
Measles virus 67.244830
dtype: float64
```

Note that only a subset of taxaons are shown in the pie chart that is those that cover at least 1% of the total reads. Others are put together and labelled “others”

A more interactive plot can be obtained using Krona:

```
from sequana import KrakenResults, sequana_data
test_file = sequana_data("test_kraken.out", "testing")
k = KrakenResults(test_file, verbose=False)
k.to_js(onweb=False) # The output filename is krona.html by default
```

Out:
An example is available in Krona example

**Total running time of the script:** (0 minutes 0.919 seconds)

Download Python source code: plot_kraken.py

Download IPython notebook: plot_kraken.ipynb

### Running median example

Plot running median on a data set

```python
from sequana.running_median import RunningMedian
from pylab import *

N = 1000
X = linspace(0, N-1, N)
```
# Create some interesting data with SNP and longer over represented section.
data = 20 + randn(N) + sin(X*2*pi/1000.*5)
data[300:350] += 10
data[500:505] += 100
data[700] = 1000

plot(X, data, "k", label="data")
rm = RunningMedian(data, 101)
plot(X, rm.run(), 'r', label="median W=201")

from sequana.stats import moving_average as ma
plot(X[100:-100], ma(data, 201), 'g', label="mean W=201")
grid()
legend()
ylim([10, 50])

Total running time of the script: (0 minutes 0.121 seconds)
Download Python source code: plot_running_median.py
Download IPython notebook: plot_running_median.ipynb

Case Examples

Effect of the trimming on SNPs detection

Description Effect of trimming (or not trimming) on the the SNPs detection.

In this case example, we will take a paired-end data set, and apply the quality pipeline using trimming quality (removing bases with quality below 30). Then, we will run the variant calling pipeline to perform the mapping on a reference and detect SNPs.

We will repeat this analysis without trimming low quality reads at all.

We will finally compare the two sets of SNPs showing that the trimming quality is not important in this example. Meaning that the mapping tool used (freebayes) is able to cope with low quality reads.

The data

We will use a paired-end data set (MiSeq 250bp). It contains 250,000 reads (X2). The organism sequenced is *Bordetella*. As a reference, we use the ENA accession *CP010347.1*. The data will be posted later but the original data were generated at Pole Biomics (Institut Pasteur) and named *Tohama-R0_S4_L001_R1_001* from which we used only the first 250,000 reads.

Here is a boxplot of the base quality across the reads showing that the quality is quite high and falls below 30 after 200 bases.
Quality pipeline

Assuming DATA (fastq.gz files) are in <DIR1> directory, type this command to create the quality pipeline and config file automatically:

```
sequana --pipeline quality --input-dir <DIR1> --project trimming
```

Then go to the project and execute the pipeline:

```
cd trimming
snakemake -s quality.rules --stats report/stats.txt -p -j 4 --forceall
```

The final cleaned reads are in trimming/report/ (referred to <DIR2> hereafter) and named after the project: (trimming_R1_.cutadapt.fastq.gz and trimming_R2_.cutadapt.fastq.gz). These two files should be used later as the input of the variant_calling pipeline, as shown hereafter.

There is no adapters in the data so in the config file, the adapter sections are empty (no forward or reverse adapters). Note, however, that bad quality bases below 30 (default) are removed. In order to set the quality to another values, use sequana with the --quality option

See also:

See the Tutorial and User guide and reference sections for more details.

Variant analysis

The output of the quality pipeline will be the input of the variant calling pipeline:

```
sequana --pipeline variant_calling --input-dir <DIR2> --project variant_trimming
```
Here you need to make sure that the config.yaml configuration file has the correct reference. See the Tutorial section (variant section).

```python
reference = "CP010347"
from bioservices import EUtils
eu = EUtils()
data = eu.EFetch(db="nucore", id=reference, rettype="gbwithparts",
    retmode="text")
with open("data.gbk", "w") as fout:
    fout.write(data.decode())

from bioservices import ENA
ena = ENA()
data = ena.get_data(reference', 'fasta')
with open("data.fa", "w") as fout:
    fout.write(data.decode())

from sequana import snpeff
v = snpeff.SnpEff("data.gbk")
```

Edit the config.yaml to change those sections:

```yaml
# snpEff parameter
snpeff:
    do: yes
    reference: "data.gbk"

# Bwa parameter for reference mapping
bwa_ref:
    reference: "data.fa"
```

Run the analysis:

```bash
cd variant_trimming
snakemake -s variant_calling.rules --stats report/stats.txt -p -j 4 --forceall
```

Once done, you should have VCF files in variant/report/ named cutadapt.ann.vcf

### No trimming

Repeat the previous two steps. In the first step, change the adapter section (cutadapt) to set the quality to zero (this prevents the trimming of bad quality bases):

```yaml
cutadapt:
    quality: 0,0
```

Change the project name e.g. no_trimming as a tag to the project in the first step and variant_no_trimming.

### SNPs results comparison

You should now have two VCF files. Here below we plot the read depth versus strand balance. The color will indicates the overall freebayes score (normalised by the largest score). A good candidate should have large score and balance value around 0.5. The y-axis shows the read depth.

```python
from pylab import *
from sequana import vcf_filter
```
In this figure the LHS (trimming) 294 SNPs were found while in the RHS (no trimming) 309 were found. The additional SNPs all have low coverage below 20. A third of them have low balance strand.

There is one SNP found in the trim case not found in no_trim. However, it is marginal with strand balance of 0.12, depth of 11, frequence of 0.73 and one of the lowest score

**Conclusions**

The detection of SNPs does not suffer from not trimming low quality bases below 30. Actually, some new SNPs are found. However, the are usually not significant (low depth, low score or unbalanced). Interestingly, the distribution of
the SNPs in the depth vs strand balance plane seems to be more centered on strand balance=0.5 could be interesting to extend the analysis to more data, lower quality, or higher quality threshold.

**Developer guide**

In this section we first look at how to include a new module (snakemake rule) in Sequana. Then, we will create a new pipeline that uses that single rule.

The rule simply counts the number of reads in a fastq file. The pipeline will only contains that unique rule.

**How to write a new module**

A *Module* (in Sequana parlance) is a directory with a set of files: a Snakemake file (also known as *Snakefile*), a README file for the documentation and a configuration file (optional). The Snakefile may be a simple snakemake rule or a set of them (a pipeline).

**Find a valid name**

All rules and pipelines must have a unique name in Sequana. We can quickly check that a name is not used in Sequana using:

```python
>>> import sequana
>>> "count" in sequana.modules.keys()
False
```

So, let us name it **count**

**Create a Snakefile rule**

A possible code that implements the **count** rule is the following Snakefile:

```python
from sequana import sequana_data
filename = sequana_data("Hm2_GTGAAA_L005_R1_001.fastq.gz", "data")

rule count:
  input: filename
  output: "count.txt"
  run:
    from sequana import FastQ
    def count(fastq):
      return len(FastQ(fastq))
    results = dict([(filename, count(filename)) for filename in input])
    with open(output[0], "w") as fout:
      fout.write("%s" % results)
```

This is not a tutorial on Snakemake but let us quickly explain this Snakefile. The first two lines use Sequana library to provide the *filename* as a test file.

Then, the rule itself is defined on line 4 where we define the rule named: **count**. We then provide on line 5 and 6 the expected input and output filenames. On line 7 onwards, we define the actual function that counts the number of reads and save the results in a TXT file.
You can now execute the Snakefile just to check that this rule works as expected:

```
snakemake -s Snakefile -f
```

You can check that the file `count.txt` exists.

**Note:** The option `-f` forces snakemake to run the rules (even though it was already computed earlier).

---

### Store the rule in a Sequana module

We now store this Snakefile in the proper place. All modules are placed either in `/sequana/pipelines` or in `/sequana/rules` directory. The tree structure looks like:

```
./
| - rules
|   | - count
|   |   | - count.rules
|   |   | - README.rst
| - pipelines
|   | - count_pipeline
|   |   | - count_pipeline.rules
|   |   | - README.rst
```

We have created a `count` directory in `/rules` and put the Snakefile in it (named `count.rules`).

A few comments: (1) there is a unique file named `config.yaml` at the root, (2) directory names must match the rule filename contain in it (3) the Snakefiles all end in `.rules`, and (4) a `README.rst` must be present in all directories.

The README file in the rules can be empty. However, the README in the pipelines’s directory is used in the documentation and automatically parsed. See *The pipeline README file* section for further details.

The `count` rules is now part of the library, which can be checked using the same code as before:

```
>>> import sequana

>>> "count" in sequana.modules.keys()
True
```

---

### Convention to design a rule

#### Use variables

Consider this Snakefile:

```python
rule bedtools_genomecov:
    input:
        __bedtools_genomecov__input
    output:
        __bedtools_genomecov__output
    params:
        options = config["bedtools"]['options']
    shell:
        bedtools genomecov {params.options} -ibam {input} > {output}
```

---

2.7. Developer guide
We tend to not hard-code any filename. So the input and output are actually variables. The variable names being the name of the rule with leading and trailing doubled underscores followed by the string `input` or `output`.

**Note:** The is a big advantage of designing rules with variables only: rules can be re-used in any pipelines without changing the rule itself; only pipelines will be different.

### Use a config file

We encourage developers to NOT set any parameter in the params section of the Snakefile. Instead, put all parameters required inside the `config.yaml` file. Since each rule has a unique name, we simply add a section with the rule name. For instance:

```yaml
bedtools_genomecov:
  options: ''
```

This is a YAML formatted file. Note that there is no information here. However, one may provide any parameters understood by the rule (here `bedtools genomecov` application) in the `options` field.

We encourage developers to put as few parameters as possible inside the config. First to not confuse users and second because software changes with time. Hard coded parameter may break the pipeline. However having the `options` field allows users to use any parameters.

**See also:**

Sequana contains many pipelines that can be used as examples. See github repo

### Add documentation in the rule

In **sequana**, we provide a sphinx extension to include the inline documentation of a rule:

```
.. snakemakerrule:: rule_name
```

This searches for the rule docstring, and includes it in your documentation. The docstring should be uniformised across all rules and pipelines. Here is our current convention:

```yaml
rule cutadapt:
  """Cutadapt (adapter removal)

Some details about the tool on what is does is more than welcome.

Required input:
  - __cutadapt__input_fastq

Required output:
  - __cutadapt__output

Required parameters:
  - __cutadapt__fwd: forward adapters as a file, or string
    - __cutadapt__rev: reverse adapters as a file, or string

Required configuration:
  .. code-block:: yaml

    cutadapt:
      fwd: "%(adapter_fwd)s"
```

38 Chapter 2. User guide and reference
How to write new pipelines

There are many rules already available in Sequana. You can easily add rules as follows:

```python
from sequana import snaketools as sm
include: sm.module['rulegraph']
```

Use sequana.snaketools

Assuming there will be a config file named `config.yaml`, the pipeline should be written as follows:

The pipeline README file

This is a WIP in progress but here is an example for the previous pipeline:

```ini
:Overview: Counts the reads in a fastq file
:Input: FastQ raw data file
:Output:
   - count.txt

Usage
~~~~~~
::
sequana init pipeline_count
snakemake -s pipeline_count.rules -f
```

Note: the README uses Restructured syntax (not markdown)
Documenting the configuration file

The configuration should be in YAML format. You should comment top-level sections corresponding to a rule as follows:

```
# This means a # character followed by a space and then
# the docstring. The first line made of #### will be removed
# and is used to make the documentation clear. No spaces
# before the section (count:) here below.
# count:  # you can add an overview
    - item1: 1  # you can add comment for an item
    - item2: 2  # you can add comment for an item
```

If valid, the block comment is interpreted and a tooltip will appear in Sequanix.

Further coding conventions

To print debugging information, warnings or more generally information, please do not use the print() function but the logger:

```
from sequana import logger
logger.debug("test")
logger.info("test")
logger.warning("test")
logger.error("test")
logger.critical("test")
```

Testing with pytest

We moved from nosetests to pytest. This framework is slightly more flexible but the main reason to move was to be able to test Qt application.

In order to run the test locally, you will need to use:

```
pip install pytest pytest-cov pytest-qt pytest-xdist pytest-mock pytest-timeout
```

Then, you can type for instance:

```
pytest -v -- durations=10 test/ --cov=sequana --cov-report term --timeout 300 -n 2
```

Here, -n 2 requires two CPUs to run the test. The option durations=10 means show the 10 longest tests.

If you want to test a single file (e.g. test_pacbio):

```
        cd test
        pytest test_pacbio.py --cov sequana.pacbio --cov-report term-missing
```
Applications (standalone)

Contents

• Applications (standalone)
  – Sequanix: GUI for snakemake workflows
  – sequana
  – sequana_coverage
  – sequana_summary
  – sequana_mapping
  – sequana_taxonomy
  – fastq related: fastq_count
  – fastq related: fastq_head
  – sequana_compressor

Sequanix: GUI for snakemake workflows

Overview a Graphical User Interface (GUI) for Sequana pipelines and any Snakemake-based workflows.

Status Production

Name sequanix

This GUI can be used to load Snakefile and their configuration file. A working directory has to be set. Once done, the configuration file can be changed in the GUI. Finally, one can run the snakefile and see the progress. Tooltips are automatically created from the configuration file (if documented).

Since snakemake has the ability to run jobs locally or on a cluster, this application can also be run either locally or a distributed computing platform (e.g., cluster with slurm scheduler). Of course, this means you can use a X environment on your cluster (ssh -X should do it).

Just type sequanix in a shell.

Note: tested under Linux only. However, Mac and Windows users should be able to use it since it is based on Python and PyQt. Again, we strongly advice to use Anaconda to install all required dependencies

Here is a snapshot.
Overview  Creates project(s) to run a Sequana pipeline(s)

The sequana executable can be used to create pipelines (and associated config file). For example:
sequana --pipeline quality --file1 R1.fastq.gz --file2 R2.fastq.gz --project TEST will create a directory called TEST with a few files such as quality.rules, config.yaml, a runme.sh and a README file. Valid pipelines can be found using:

```
sequana --show-pipelines
```

There are many more options and documentation. Please use the --help option for more information.

**sequana_coverage**

**Description**  Show coverage and interval of confidence to identify under and over represented genomic regions.

**Help**  please use sequana_coverage --help

**Docker**

```
git pull sequana/sequana_coverage
```

See github sequana_coverage docker page for details

**Sequana**  See GenomeCov to use the coverage in your own script.

**Gallery**  See examples in the gallery

Starting from a BED file and its reference, one can use this command in a shell:

```
sequana_coverage --input JB409847.sorted.bed -o
    --reference JB409847.fa --show-html
```

It creates an HTML report with various images showing the coverage and GC versus coverage plots. It also provides a set of CSV files with low or high coverage regions (as compared to the average coverage).

**See also:**

the underlying algorithm is described in details in the documentation (sequana.bedtools.GenomeCov).

**sequana_summary**

**Description**  Prints basic statistics about a set of NGS input files. Currently handles Fastq (gzipped or not) or BED files (coverage).

**sequana_mapping**

**Description**  a simple application to map reads onto a genome given one or two FastQ files (gzipped) and a reference.

**sequana_taxonomy**

**Description**  Creates a HTML document with Krona and pie chart of taxonomic content of a set of FastQ files. Uses Kraken and a dedicated Sequana database.
fastq related: fastq_count

**Description**  count number of reads and lines

Example:
```
fastq_count --input test.fastq.gz
```

fastq related: fastq_head

**Description**  Extract head of a fastq files (zipped or not)

Example:
```
fastq_head --input input.fastq.gz --nlines 10000 --output output.fastq.gz
```

sequana_compressor

**Description**  standalone on top of the compressor pipeline to compress/decompress FastQ files in different formats, recursively and using multithreaded and multicore tools.

Please see Compressor for details.

References

Contents

- References
  - Adapters
  - BAMTOOLS related
  - BEDTOOLS related (coverage)
  - Coverage (theoretical)
  - Access to online database (e.g. ENA)
  - Experimental design
  - FASTQ module
  - Kmer module
  - IOTools module
  - Taxonomy related (Kraken - Krona)
  - Pacbio module
  - Phred quality
  - Running median
  - Snakemake module
Adapters

Utilities to manipulate adapters

Adapters removal can be performed by many different tools such as CutAdapt, AlienTrimmer, Trimmomatic. Unfortunately, they tend to use different formats from FASTA to text files. Moreover outputs are generally also reported in different formats.

Tools to extract specific adapters from FASTA files would also be handy. For instance, you may have all your adapters in a single file.

In this module, we provide:

- tools to manipulate adapters stored in Fasta format (*AdapterReader*).
- tools to export Fasta files with adapter content into other formats required by various adapter removal software
- A tool used to extract adapters from a FASTA file given their identifier, or sequence *FindAdaptersFromDesign*.

Our convention is to store list of adapters in FASTA format, which can be read using the *AdapterReader*:

```python
from sequana import sequana_data, AdapterReader
filename = sequana_data("adapters_Nextera_fwd.fa")
ar = AdapterReader(filename)
ar.get_adapter_by_index("N501")
```

Given a design file (see mod:sequana.expdesign), and a name for the type of adapters, one can easily extract the subset of relevant adapters to be used for a sample. Currently, the following set of adapters/design are available:

- Nextera single and double indexing
- Rubicon single indexing
- PCRFree single indexing

For instance given a design file that gives the mapping between samples and a set of Nextera adapters, one would use:

```python
>>> from sequana import *
>>> filename = sequana_data("test_expdesign_hiseq.csv")
>>> design = ExpDesignAdapter(filename)
>>> fa = FindAdaptersFromDesign(design, "PCRFree")
>>> print(fa.sample_names[0])
'553-iH2-1'
>>> fa.get_adapters_from_sample("553-iH2-1")
```

See *FindAdaptersFromDesign* for details.

```python
class Adapter (identifier, sequence=None, comment=None)
    Class to store one adapter

    An adapter is just a sequence from a FASTA file. It contains an identifier, a sequence and possibly a comment.
```

2.9. References 45
Warning: The identifier provided must not contain the starting ">" character, which is added automatically when needed.

One can check if an adapter is equal to another. Only the sequence is checked for equality though.
Some Sequana notation have been added in the identifier to ease retrieval of index’s name and index’s sequence:

```
>NextFlex_PCR_Free_adapter1|name:1|seq:CGATGT
```

Of course the string CGATGT must be found in the sequence itself.

```
ar = AdapterReader(sequana_data("adapters_PCRFree_fwd.fa"))  
adapter = Adapter(ar[0])  
adapter.identifier  
adapter.comment  
adapter.index_sequence  
adapter.sequence  
adapter.name
```

- **comment**: R/W adapter’s identifier
- **identifier**: R/W adapter’s identifier
- **index_sequence**: Read only access to the index sequence
- **name**: Read only access to the index name
- **sequence**: R/W adapter’s sequence

**Class:** 

```python
class AdapterReader (filename)  
Reader of FASTA file dedicated to adapters  
```

A Fasta is just a set of this kind of paired-lines:

```
>Nextera_index_N501|name:N501|seq:ACGT optional comment  
ACGTACGTACGT
```

where the **optional comment** is separated from the identifier by a tabulation.

In the FASTA identifier, the first pipe delimits the official name (left hand side) from the name tag. The information on this example may be redundant but the **name** will be used throughout the Sequana code to ensure reproducibility.

**Note:** sequences are all in big caps.

**Note:** the universal adapter has no index so does not need to have the any tags for the name of index sequence. However, it must be called **Universal_Adapter**

```
>>> from sequana import sequana_data, AdapterReader  
>>> filename = sequana_data("adapters_Nextera_fwd.fa")
```
>>> ar = AdapterReader(filename)
>>> candidate = ar.get_adapter_by_index_name("S505")
>>> print(candidate[0])
>Nextera_index_S505|name:S505|seq:GTAAGGAG
AATGATACGGCGACCACCGAGATCTACACGTAAGGAGTCGGCAGCGTC
>>> len(ar)
56

Note: Checks for uniqueness of the identifiers. It not unique, an error is raised

Sources  document illumina #1000000002694 v01


Constructor

Parameters filename (str) – the input FASTA file

comments

get_adapter_by_identifier (text)
Return adapter whose identifier matches the user text

Parameters index_identifier – the unique index identifier to be found. If several sequence do match, this is an error meaning the fasta file with all adapters is not correctly formatted.

Returns the adapter that match the index_name (if any) otherwise returns None

get_adapter_by_index_name (index_name)
Return adapter for the index name provided

Can be used only if the identifier contains the tag:

|name:an_index_to_be_found

For instance:

>>Nextera_blabal|name:N505|seq:ACGT
>>Nextera_blabal|seq:ACGT|name:N505
>>Nextera_blabal|name:N505|seq:ACGT

are valid identifiers

get_adapter_by_index_seq (index_name)
See get_adapter_by_index_name().

get_adapter_by_sequence (subsequence)
Return one or several adapters with sub-sequence in their sequence

Parameters subsequence (str) – a string (ACGT letters)

Returns name and sequence in FASTA format that have the user sequence contained in their sequence

2.9. References
If the subsequence is short, it may return more than 1 adapters. Besides, the sequence is searched for without position information right now.

**identifiers**

**index_names**

**index_sequences**

**reverse()**

Reverse all sequences inplace

```python
>>> from sequana import sequana_data, AdapterReader
>>> filename = sequana_data("adapters_Nextera_fwd.fa")
>>> filename2 = sequana_data("adapters_Nextera_rev.fa")
>>> ar = AdapterReader(filename)
>>> ar2 = AdapterReader(filename2)
>>> ar.reverse()
>>> ar == ar2
True
```

**reverse_complement()**

Reverse-complement all sequences inplace

```python
>>> from sequana import sequana_data, AdapterReader
>>> filename = sequana_data("adapters_Nextera_fwd.fa")
>>> filename2 = sequana_data("adapters_Nextera_revcomp.fa")
>>> ar = AdapterReader(filename)
>>> ar.reverse_complement()
>>> ar.to_fasta()
>>> ar == ar2
```

**sequences**

**to_dict()**

Returns dictionary with key as identifier and values as list with comments and sequences

**tofasta (filename)**

Save sequences into fasta file

**class FindAdaptersFromDesign (design_filename, adapters)**

Extract adapter(s) corresponding to an experimental design file

Used by sequana main script to build the adapter files for multi-samples projects as input to various adapter removal software.

**Constructor**

Parameters

- **design_filename (str)** – a CSV file that is compatible with our `sequana.expdesign.ExpDesignAdapter`

- **adapters** – the type of adapters (PCRFree or Nextera, or Rubicon)

The files of adapters are stored in Sequana and accessible with the `sequana_data` function. So, for instance if adapters is set to Nextera, the following file is used to identify the adapters:

`sequana_data("adapters_Nextera_fwd.fa")`
New adapters files can be added on request. Currently, Nextera and PCRFree are available. Rubicon and TruSeq will be added soon.

check()

get_adapters_from_sample(sample_name, include_universal=True, include_transposase=True)

Return a dictionary with adapters corresponding to the sample name

Parameters

- **sample_name** (*str*) – a valid sample name as found in the design file. One can check the content of the **sample_names** attribute.
- **include_transposase** (*bool*) – include transposase if found
- **include_universal** (*bool*) – include universal adapter if found

Returns a dictionary with the adapters in forward, reverse, reverse complement for index1 and index2 (if relevant).

get_sample(sample_name)

Return basic info about the sample name (from the design file)

sample_names

return all sample names contained in the design file

save_adapters_to_fasta(sample_name, include_universal=True, include_transposase=True, output_dir='. ')

Get index1, index2 and universal adapter

adapter_removal_parser(filename)

Parses output of AdapterRemoval software

```python
>>> from sequana import adapters, sequana_data

>>> data = sequana_data("test_adapter_removal_output.txt", "testing")

>>> results = adapters.adapter_removal_parser(data)

>>> results['adapter1']
'AGATCGGAAGAGCACACGTCTGAACTCCAGTCACNNNNNNATCTCGTATGCCGTCTTCTGCTTG'
```

adapters_to_clean_ngs(input_filename, output_filename='adapters_nsg.txt')

Convert a FASTA formatted file into clean_nsg format

Parameters

- **input_filename** (*str*) – the input FASTA file
- **output_filename** (*str*) – a TSV formatted file

fasta_fwd_rev_to_columns(file1, file2=None, output_filename=None)

From 2 FASTA files (reverse and forward) adapters, returns 2-columns file

This is useful for some tools related to adapter removal that takes as input this kind of format

Parameters

- **filename1** (*str*) – FASTA format
- **filename2** (*str*) – FASTA format (optional)

The files must have a one-to-one mapping

get_sequana_adapters(type, direction)

Return path to a list of adapters in FASTA format

2.9. References
• **tag** – PCRFree, Rubicon, Nextera
• **type** – fwd, rev, revcomp

Returns path to the adapter filename

**BAMTOOLS related**

Tools to manipulate BAM/SAM files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Alignment</code></td>
<td>Helper class to retrieve info about Alignment</td>
</tr>
<tr>
<td><code>BAM(filename[, mode])</code></td>
<td>BAM data structure</td>
</tr>
<tr>
<td><code>SAMFlags([value])</code></td>
<td>Utility to extract bits from a SAM flag</td>
</tr>
</tbody>
</table>

**Note:** BAM being the compressed version of SAM files, we do not implement any functionalities related to SAM files. We strongly encourage developers to convert their SAM to BAM.

```python
from sequana import BAM, sequana_data
b = BAM(sequana_data("test.bam"))
len(b)
1000
```

**Note:** Once you loop over this data structure, you must call `reset()` to force the next iterator to start at position 0. The methods implemented in this data structure take care of that for you thanks to a decorator called seek. If you want to use the `next()` function, call `reset()` to make sure you start at the beginning.

```python
bam_analysis_to_json(filename)
```

Create a json file with information related to the bam file.

This includes some metrics (see `get_stats()`; eg MAPQ), combination of flags, SAM flags, counters about the read length.

```python
get_flags()
```

Return flags of all reads as a list

**See also:**

```python
get_flags_as_df(), SAMFlags
```

```python
get_flags_as_df()
```

Returns flags as a dataframe
>>> from sequana import BAM, sequana_data
>>> b = BAM(sequana_data('test.bam'))
>>> df = b.get_flags_as_df()
>>> df.sum()
1    1000
2     484
4      2
8      2
16     499
32     500
64     477
128    523
256    64
512     0
1024    0
2048    0
dtype: int64

See also:

SAMFlags for meaning of each flag

get_full_stats_as_df()
Return a dictionary with full stats about the BAM file
The index of the dataframe contains the flags. The column contains the counts.

>>> from sequana import BAM, sequana_data
>>> b = BAM(sequana_data("test.bam"))
>>> df = b.get_full_stats_as_df()
>>> df.query("description=='average quality'")
36.9

Note: uses samtools behind the scene

gc_content()
Return GC content for all reads (mapped or not)

See also:

plot_gc_content()
g_length_count()
Return counter of all fragment lengths

g_mapped_read_length()
Return dataframe with read length for each read

g_mapq_as_df()
Return dataframe with mapq for each read

g_metrics_count()
Count flags/mapq/read length in one pass. It is the faster and less costly method to have these interesting metrics.

g_read_names()
Return the reads’ names
get_samflags_count()
Count how many reads have each flag of sam format after count metrics.

get_stats()
Return basic stats about the reads

Returns
dictionary with basic stats:
• total_reads : number reads
• mapped_reads : number of mapped reads
• unmapped_reads : number of unmapped
• mapped_proper_pair : R1 and R2 mapped face to face
• hard_clipped_reads: number of reads with supplementary alignment
• reads_duplicated: number of reads duplicated

Warning: works only for BAM files. Use get_full_stats_as_df() for SAM files.

iter_mapped_reads()
Return an iterator on the reads that are mapped

iter_unmapped_reads()
Return an iterator on the reads that are unmapped

plot_bar_flags(logy=True, fontsize=16, filename=None)
Plot an histogram of the flags contained in the BAM

```python
from sequana import BAM, sequana_data
b = BAM(sequana_data('test.bam', "testing"))
b.plot_bar_flags()
```
See also:

SAMFlags for meaning of each flag

plot_bar_mapq (fontsize=16, filename=None)
Plots bar plots of the MAPQ (quality) of alignments

```python
from sequana import BAM, sequana_data
b = BAM(sequana_data('test.bam', "testing"))
b.plot_bar_mapq()
```
plot gc content (fontsize=16, ec='k', bins=100)
plot GC content histogram

Params bins a value for the number of bins or an array (with a copy() method)

Parameters ec – add black contour on the bars

```python
from sequana import BAM, sequana_data
b = BAM(sequana_data('test.bam'))
b.plot_gc_content()
```
to_fastq(filename)
Export the BAM to FastQ format

**Warning:** to be tested

**Todo**
comments from original reads?

Method 1 (bedtools):

```
bedtools bamtofastq -i JB409847.bam -fq test1.fastq
```

Method 2 (samtools):

```
samtools bam2fq JB409847.bam > test2.fastq
```

Method 3 (sequana):

```
from sequana import BAM
BAM(filename)
BAM.to_fastq("test3.fastq")
```

Note that the samtools method removes duplicated reads so the output is not identical to method 1 or 3.
class **Alignment** *(alignment)*  
Helper class to retrieve info about Alignment

Takes an alignment as read by `BAM` and provides a simplified version of `pysam.Alignment` class.

```python
>>> from sequana.bamtools import Alignment
>>> from sequana import BAM, sequana_data

>>> b = BAM(sequana_data("test.bam"))
>>> segment = next(b)
>>> align = Alignment(segment)
>>> align.as_dict()
```

The original data is stored in hidden attribute `_data` and the following values are available as attributes or dictionary:

- **QNAME**: a query template name. Reads/segment having same QNAME come from the same template. A QNAME set to `*` indicates the information is unavailable. In a sam file, a read may occupy multiple alignment
- **FLAG**: combination of bitwise flags. See `SAMFlags`
- **RNAME**: reference sequence
- **POS**
- **MAPQ**: mapping quality if segment is mapped. equals -10 log10 Pr
- **CIGAR**:
- **RNEXT**: reference sequence name of the primary alignment of the NEXT read in the template
- **PNEXT**: position of primary alignment
- **TLEN**: signed observed template length
- **SEQ**: segment sequence
- **QUAL**: ascii of base quality

**constructor**

Parameters **alignment** – alignment instance from `BAM`

```python
as_dict()  
```

class **SAMFlags** *(value=4095)*  
Utility to extract bits from a SAM flag

```python
>>> from sequana import SAMFlags

>>> sf = SAMFlags(257)
>>> sf.get_flags()
[1, 256]
```

You can also print the bits and their description:

```python
print(sf)
```
### Sequana, Release 0.1.21

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>template having multiple segments in sequencing</td>
</tr>
<tr>
<td>2</td>
<td>each segment properly aligned according to the aligner</td>
</tr>
<tr>
<td>4</td>
<td>segment unmapped</td>
</tr>
<tr>
<td>8</td>
<td>next segment in the template unmapped</td>
</tr>
<tr>
<td>16</td>
<td>SEQ being reverse complemented</td>
</tr>
<tr>
<td>32</td>
<td>SEQ of the next segment in the template being reverse complemented</td>
</tr>
<tr>
<td>64</td>
<td>the first segment in the template</td>
</tr>
<tr>
<td>128</td>
<td>the last segment in the template</td>
</tr>
<tr>
<td>256</td>
<td>secondary alignment</td>
</tr>
<tr>
<td>512</td>
<td>not passing filters, such as platform/vendor quality controls</td>
</tr>
<tr>
<td>1024</td>
<td>PCR or optical duplicate</td>
</tr>
<tr>
<td>2048</td>
<td>supplementary alignment</td>
</tr>
</tbody>
</table>


**get_flags**

Return the individual bits included in the flag

**get_meaning**

Return all description sorted by bit

### BEDTOOLS related (coverage)

Utilities for the genome coverage

```python
class GenomeCov(input_filename, genbank_file=None, low_threshold=.3, high_threshold=3, ldt=0.5, hdt=0.5)
```

Create a list of dataframe to hold data from a BED file generated with samtools depth.

This class can be used to plot the coverage resulting from a mapping, which is stored in BED format. The BED file may contain several chromosomes. There are handled independently and accessible as a list of `ChromosomeCov` instances.

Example:

```python
from sequana import GenomeCov, sequana_data

filename = sequana_data('JB409847.bed')
reference = sequana_data("JB409847.fasta")

gencov = GenomeCov(filename)
gencov.compute_gc_content(reference)

gencov = GenomeCov(filename)
for chrom in gencov:
    chrom.running_median(n=3001, circular=True)
    chrom.compute_zscore()
    chrom.plot_coverage()

gencov[0].plot_coverage()
```

2.9. References
Results are stored in a list of `ChromosomeCov` named `chr_list`.

**constructor**

**Parameters**

- **input_filename** *(str)* – the input data with results of a bedtools genomecov run. This is just a 3-column file. The first column is a string (chromosome), second column is the base position and third is the coverage.

- **genbank_file** *(str)* – annotation file of your reference.

- **low_threshold** *(float)* – threshold used to identify under-covered genomic region of interest (ROI). Must be negative

- **high_threshold** *(float)* – threshold used to identify over-covered genomic region of interest (ROI). Must be positive

- **ldtr** *(float)* – fraction of the low_threshold to be used to define the intermediate threshold in the double threshold method. Must be between 0 and 1.

- **rdtr** *(float)* – fraction of the low_threshold to be used to define the intermediate threshold in the double threshold method. Must be between 0 and 1.

**circular**

Get the circularity of chromosome(s). It must be a boolean.
**compute_gc_content** *(fasta_file, window_size=101, circular=False, letters=['G', 'C', 'c', 'g'])*

Compute GC content of genome sequence.

**Parameters**

- **fasta_file** *(str)* – fasta file name.
- **window_size** *(int)* – size of the sliding window.
- **circular** *(bool)* – if the genome is circular (like bacteria chromosome)

Store the results in the **ChromosomeCov.df** attribute (dataframe) with a column named `gc`.

**feature_dict**

Get the features dictionary of the genbank.

**gc_window_size**

Get or set the window size to compute the GC content.

**genbank_filename**

Get or set the genbank filename to annotate ROI detected with **ChromosomeCov.get_roi()**. Changing the genbank filename will configure the **GenomeCov.feature_dict**.

**get_stats** *(output='json')*

Return basic statistics for each chromosome

**Returns** dictionary with chromosome names as keys and statistics as values.

**See also:** **ChromosomeCov**.

**hist** *(logx=True, logy=True, fignum=1, N=20, lw=2)*

**to_csv** *(output_filename, **kwargs)*

Write all data in a csv.

**Parameters**

- **output_filename** *(str)* – csv output file name.
- **kwargs** *(**dict)* – parameters of pandas.DataFrame.to_csv().

**window_size**

Get or set the window size to compute the running median. Size must be an interger.

**class ChromosomeCov** *(df, genomecov, thresholds=None)*

Factory to manipulate coverage and extract region of interests.

**Example:**

```python
from sequana import GenomeCov, sequana_data
filename = sequana_data("virus.bed")

gencov = GenomeCov(filename)

chrcov = gencov[0]
chrcov.running_median(n=3001)
chrcov.compute_zscore()
chrcov.plot_coverage()
df = chrcov.get_roi().get_high_roi()
```
The `df` variable contains a dataframe with high region of interests (over covered)

See also:
sequana_coverage standalone application

**constructor**

**Parameters**

- `df` – dataframe with position for a chromosome used within `GenomeCov`. Must contain the following columns: ["chr", "pos", "cov"]

- `thresholds` – a data structure `DoubleThresholds` that holds the double threshold values.

**bed**

**columns()**

Return immutable ndarray implementing an ordered, sliceable set.

**compute_zscore**(*k=2, step=10, use_em=True, verbose=True*)

Compute zscore of coverage and normalized coverage.

**Parameters**

- `k (int)` – Number gaussian predicted in mixture (default = 2)
• **step** (*int*) – (default = 10). This parameter is used to speed up computation and is ignored if the length of the coverage/sequence is below 100,000.

Store the results in the `df` attribute (dataframe) with a column named `zscore`.

**Note:** needs to call `running_median()` before hand.

**get_centralness()**
Proportion of central (normal) genome coverage
This is 1 - (number of non normal data) / (total length)

**Note:** depends on the thresholds attribute being used.

**get_cv()**
Return the coefficient variation
The coefficient of variation (CV) is defined as sigma / mu
To get percentage, you must multiply by 100.

**get_df()**

**get_evenness()**
Return Evenness of the coverage

**Reference** Konrad Oexle, Journal of Human Genetics 2016, Evaluation of the evenness score in NGS.

work before or after normalisation but lead to different results.

**get_gaussians()**

**get_gc_correlation()**
Return the correlation between the coverage and GC content
The GC content is the one computed in `GenomeCov.compute_gc_content()` (default window size is 101)

**get_max_gc_correlation**(reference)
Plot correlation between coverage and GC content by varying the GC window
The GC content uses a moving window of size W. This parameter affects the correlation between coverage and GC. This function find the optimal window length.

**get_mean_cov()**

**get_roi()**
Keep positions with zscore outside of the thresholds range.

**Returns** a dataframe from `FilteredGenomeCov`

**Note:** depends on the thresholds low and high values.

**get_size()**
**get_stats** (output='json')

Return basic stats about the coverage data

**get_var_coef()**

**moving_average (n, circular=False)**

Compute moving average of the genome coverage

**Parameters**

- **n** – window’s size. Must be odd
- **circular** (bool) – is the chromosome circular or not

Store the results in the `df` attribute (dataframe) with a column named `ma`.

**plot_coverage** (filename=None, fontsize=16, rm_lw=1, rm_color='#0099cc', rm_label='Running median', th_lw=1, th_color='r', th_ls='--', main_color='k', main_lw=1, main_kwargs={})

Plot coverage as a function of base position.

**Parameters**

- **filename** – In addition, the running median and coverage confidence corresponding to the lower and upper zscore thresholds

**Note:** uses the thresholds attribute.

**plot_gc_vs_coverage** (filename=None, bins=None, Nlevels=6, fontsize=20, norm='log', ymin=0, ymax=100, contour=True, **kwargs)

**plot_hist_coverage** (logx=True, logy=True, fontsize=16, N=20, fignum=1, hold=False, alpha=0.5, filename=None, **kw_hist)

**plot_hist_normalized_coverage** (filename=None, binwidth=0.1, max_z=4)

Barplot of the normalized coverage with gaussian fitting

**plot_hist_zscore** (fontsize=16, filename=None, max_z=6, binwidth=0.5, **hist_kargs)

Barplot of the zscore values

**running_median** (n, circular=None)

Compute running median of genome coverage

**Parameters**

- **n** (int) – window’s size.
- **circular** (bool) – if a mapping is circular (e.g. bacteria whole genome sequencing), set to True

Store the results in the `df` attribute (dataframe) with a column named `rm`.

Changed in version 0.1.21: Use Pandas rolling function to speed up computation.

**set_gaussians** (gaussians)

Set gaussians predicted if you read a csv file generated by `GenomeCov`.

**to_csv** (filename=None, start=None, stop=None, **kwargs)

Write CSV file of the dataframe.

**Parameters**

- **filename** (str) – csv output filename. If None, return string.
- **start** (int) – start row index.
• **stop** (*int*) – stop row index.

Params of `pandas.DataFrame.to_csv()`:

**Parameters**

• **columns** (*list*) – columns you want to write.
• **header** (*bool*) – determine if the header is written.
• **index** (*bool*) – determine if the index is written.
• **float_format** (*str*) – determine the float format.

```python
class DoubleThresholds(low=-3, high=3, ldtr=0.5, hdtr=0.5)
    Simple structure to handle the double threshold for negative and positive sides
    Used yb GenomeCov and related classes.
    dt = DoubleThresholds(-3,4,0.5,0.5)
```

This means the low threshold is -3 while the high threshold is 4. The two following values must be between 0 and 1 and are used to define the value of the double threshold set to half the value of the main threshold by default.

Internally, the main thresholds are stored in the low and high attributes. The secondary thresholds are derived from the main thresholds and the two ratios. The ratios are named ldtr and hdtr for low double threshold ratio and high double threshold ration. The secondary thresholds are denoted low2 and high2 are are update automatically if low, high, ldtr or hdtr are changed.

```python
copy()
get_args()
hdtr
high
high2
ldtr
low
low2
```

**Coverage (theoretical)**

```python
class Coverage(N=None, L=None, G=None, a=None)
    Utilities related to Lander and Waterman theory
    We denote $G$ the genome length in nucleotides and $L$ the read length in nucleotides. These two numbers are in principle well defined since $G$ is defined by biology and $L$ by the sequencing machine.
    The total number of reads sequenced during an experiment is denoted $N$. Therefore the total number of nucleotides is simply $NL$.
    The depth of coverage (DOC) at a given nucleotide position is the number of times that a nucleotide is covered by a mapped read.
    The theoretical fold-coverage is defined as :
    $$a = \frac{NL}{G}$$
```
that is the average number of times each nucleotide is expected to be sequenced (in the whole genome). The fold-coverage is often denoted \( aX \) (e.g., 50X).

In the \texttt{Coverage} class, \( G \) and \( N \) are fixed at the beginning. Then, if one changes \( a \), then \( N \) is updated and vice-versa so that the relation \( a = NL/G \) is always true:

```python
>>> cover = Coverage(G=1000000, L=100)
>>> cover.N = 100000 # number of reads
>>> cover.a = 10   # What is the mean coverage
>>> cover.a = 50
>>> cover.N = 500000
```

From the equation aforementionned, and assuming the reads are uniformly distributed, we can answer a few interesting questions using probabilities.

In each chromosome, a read of length \( L \) could start at any position (except the last position \( L-1 \)). So in a genome \( G \) with \( n_c \) chromosomes, there are \( G - n_c(L - 1) \) possible starting positions. In general \( G >> n_c(L - 1) \) so the probability that one of the \( N \) read starts at any specific nucleotide is \( N/G \).

The probability that a read (of length \( L \)) covers a given position is \( L/G \). The probability of not covering that location is \( 1 - L/G \). For \( N \) fragments, we obtain the probability \( (1 - L/G)^N \). So, the probability of covering a given location with at least one read is:

\[
P = 1 - \left( 1 - \frac{L}{G} \right)^N
\]

Since in general, \( N >> 1 \), we have:

\[
P = 1 - \exp^{-NL/G}
\]

From this equation, we can derive the fold-coverage required to have e.g., \( E = 99\% \) of the genome covered:

\[
a = \log(1/(E - 1))
\]

equivalent to

\[
a = -\log(1 - E)
\]

The method \texttt{get_required_coverage()} uses this equation. However, for numerical reason, one should not provide \( E \) as an argument but \((1-E)\). See \texttt{get_required_coverage()}.

Other information can also be derived using the methods \texttt{get_mean_number_contigs()}, \texttt{get_mean_contig_length()}. See \texttt{get_mean_contig_length()}.

See also:

\texttt{get_table()} that provides a summary of all these quantities for a range of coverage.


\( G \)

\text{genome length}

\( L \)

\text{length of the reads}

\( N \)

\text{number of reads defined as } aG/L
a coverage defined as NL/G

**get_mean_contig_length()**
Expected length of the contigs

\[ \frac{e^a - 1}{a} L \]

**get_mean_number_contigs()**
Expected number of contigs

A binomial distribution with parameters \( N \) and \( p \)

\[(aG/L) \exp^{-a}\]

**get_mean_reads_per_contig()**
Expected number of reads per contig

Number of reads divided by expected number of contigs:

\[ \frac{N}{N \exp^{-a}} = e^a \]

**get_percent_genome_sequenced()**
Return percent of the genome covered

\[ 100(1 - \exp^{-a}) \]

**get_required_coverage** \((M=0.01)\)
Return the required coverage to ensure the genome is covered

A general question is what should be the coverage to make sure that e.g. \( E=99\% \) of the genome is covered by at least a read.

The answer is:

\[ \log^{-1/(E-1)} \]

This equation is correct but have a limitation due to floating precision. If one provides \( E=0.99 \), the answer is 4.6 but we are limited to a maximum coverage of about 36 when one provides \( E=0.9999999999999999 \) after which \( E \) is rounded to 1 on most computers. Besides, it is no convenient to enter all those numbers. A scientific notation would be better but requires to work with \( M = 1 - E \) instead of \( E \).

\[ \log^{-1/-M} \]

So instead of asking the question what is the requested fold coverage to have \( 99\% \) of the genome covered, we ask the question what is the requested fold coverage to have \( 1\% \) of the genome not covered. This allows us to use \( M \) values as low as 1e-300 that is a fold coverage as high as 690.

**Parameters** \( M \texttt{ (float) } \) – this is the fraction of the genome not covered by any reads (e.g. 0.01 for 1%). See note above.

**Returns** the required fold coverage

2.9. References 65
# The inverse equation is required fold coverage $= \frac{\log(-1/(E - 1))}{\text{Uncovered genome}}$

**get_summary()**

Return a summary (dictionary) for the current fold coverage

**get_table(coverage=None)**

Return a summary dataframe for a set of fold coverage

**Parameters**

- **coverage (list)** – if None, coverage list starts at 0.5 and ends at 10 with a step of 0.5

## Access to online database (e.g. ENA)

Utilities to access to online FASTA, taxon, lineage ...

**class ENADownload**

Downloader to retrieve genome fasta files from ENA amongst other things

In order to facilitate the download of FASTA files (e.g. to build a Kraken DB), this class can be used to download a bunch of FASTA files, or just one given its accession.

Pre-defined lists are available from ENA. We refer to them as virus, plasmid, phage, archaealvirus, archaea, bacteria, organelle, viroid. In addition we have predefined lists within Sequana. For now, there is one named macaca_fascicularis.

**Warning:** the header of the FASTA files are changed to add the GI number instead of embl.
constructor

download_accession(acc, output='Custom')
   Download a specific FASTA file given its ENA accession number

download_archaea()

download_archaealvirus()

download_bacteria()
    organisms (may 2016)

Note: this download method is the longest to end. It took about 20mins on a good connection.

download_fasta(filelist, output_dir=None, from_ena=True)
   Download a FASTA (or list of)

   Parameters
   filelist -- a name to find on the ENA web server OR the name of an accession number.

   Warning: The filename is named after the accession without .X number If there are several variant .1, .2 the later will be used. This should not happen if the list is properly defined.

download_list()
   Download all standard lists of accession numbers from ENA

download_macaca()

download_organelle()

download_phage()

download_plasmids()

download_viroid()

download_viruses()

ena_id_to_gi_number(identifiers)

switch_header_to_gi(acc)
   Kraken will only accept the GI from NCBI so we need to convert the ENA accession to GI numbers

class EUtilsTools
   Simple wrapper around EUtils to fetch basic informatino about an accession number

>>> from sequana.databases import EUtilsTools
>>> et.accession_to_info("K01711.1")
{'K01711.1': {'accession': '331784',
               'comment': 'Measles virus (strain Edmonston), complete genome',
               'gi': '331784',
               'identifier': 'gi|331784|gb|K01711.1|MEANPCG[331784]','
               'taxid': '11234'}}

accession_to_info(ids)
   An accession or list of them returns list of dictionaries

2.9. References
**Experimental design**

Module to handle experimental design files (adapters)

Sequencers or experimentalists create so-called design files to store information about the sequencing experiment. For example the name of the samples, the sample well, and the index sequences.

The format used to store the design information may vary from one sequencing platform to another. The design file can be used for many different purposes. Currently, we only use them to retrieve information about adapters.

Since there are different formats possible, we decided on a minimal and common set of information. For now, this is a CSV file with the following minimal header:

<table>
<thead>
<tr>
<th>Sample_ID, Index1_Seq</th>
</tr>
</thead>
</table>

or, for double-indexing:

<table>
<thead>
<tr>
<th>Sample_ID, Index1_Seq, Index2_Seq</th>
</tr>
</thead>
</table>

Users should only use the *ExpDesignAdapter* class, which understands the different formats. Currently, the design file created by MiSeq Illumina machine

```python
class ExpDesignAdapter(filename, verbose=True)
```

Generic Experimental design class

This class is used to store the mapping between sample ID and adapter used.

The design information is stored as a dataframe in the attribute df.

The input format is not unique. There are currently 3 different inputs possible as defined in

- `ExpDesignGeneric`
- `ExpDesignMiSeq`
- `ExpDesignHiSeq (2500)`

The dataframe index is the list of sample identifiers (Sample_ID). The columns contain at least the following:

<table>
<thead>
<tr>
<th>Index1_Seq, Index1_ID, Index2_Seq, Index2_ID</th>
</tr>
</thead>
</table>

Example:

```python
from sequana import *
filename = sequana_data('test_test_expdesign_hiseq.csv')
eda = ExpDesignAdapter(filename)
```

**constructor**

**Parameters**

- `filename (str)` – the input design file. Can also be an instance of `ExpDesignAdapter` itself.
- `verbose (bool)` –

```python
class ExpDesignMiSeq(filename)
```

Dedicated experimental design format from Illumina MiSeq sequencers

This MiSeq design format has the following format:
In the Data section, a CSV file is to be found with the following header:

```
Sample_ID,Sample_Name,Sample_Plate,Sample_Well,I7_Index_ID,index,
Sample_Project,Description
```

The index column may be prefixed. For instance as NFXX where XX is the index so NF should be dropped.

If double-indexing, the header is:

```
Sample_ID,Sample_Name,Sample_Plate,Sample_Well,I7_Index_ID,
index,I5_Index_ID,index2,Sample_Project,Description
```

```python
filename = sequana_data("test_expdesign_miseq_illumina_1.csv")
ff = ExpDesignMiSeq(filename)
ff.df
```

class **ExpDesignHiSeq** (*filename*)

Dedicated experimental design format created by a demultiplexing soft.

This format is used by a demultiplex software used locally at biomics platform. The format of the header is:

```
FCID,Lane,SampleID,SampleRef,Index Seq, Description,Control,Recipe,Operator, Project
```

This is a format that may change in the future.

The SampleID is convert into Sample_ID. “Index Seq”. Note that “Index Seq” may be empty, or filled with an index sequence, or 2 index sequences separated by a “-” sign.

note also FCID = flowcell ID

class **ExpDesignBase** (*filename*)

The Base class for all ExpDesignAdapter classes

The input filename must be a CSV file with at least the following column in the header:

```
Sample_ID
```

Derived class must define at least **Index1_Seq** and possibly **Index2_Seq**.

Examples of specialised classes are **ExpDesignMiSeq, ExpDesignHiSeq**.

**check()**

Check the presence of the Sample_ID column

**read()**

Read a CSV file
FASTQ module

Utilities to manipulate FASTQ and Reads

**class Identifier** (identifier, version='unknown')

Class to interpret Read’s identifier

```python
>>> from sequana import Identifier
>>> ident.info['x_coordinate']
'15343'
```

Currently, the following identifiers will be recognised automatically:

**Illumina 1.4** An example is:

```text
@HWUSI-EAS100R:6:73:941:1973#0/1
```

**Illumina 1.8** An example is:

```text
@EAS139:136:FC706VJ:2:2104:15343:197393 1:Y:18:ATCACG
```

Other that could be implemented are NCBI

```text
@FSRRS4401BE7HA [length=395] [gc=36.46] [flows=800] [phred_min=0]
[phred_max=40] [trimmed_length=95]
```

Information can also be found here [http://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm](http://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm)

**class FastQ** (filename, verbose=False)

Class to handle FastQ files

Some of the methods are based on pysam but a few are also original to sequana. In general, input can be zipped or not and output can be zipped or not (based on the extension).

An example is the `extract_head()` method:

```python
f = FastQ("input_file.fastq.gz")
f.extract_head(100000, output='test.fastq')
f.extract_head(100000, output='test.fastq.gz')
```

equivalent to:

```bash
zcat myreads.fastq.gz | head -100000 | gzip > test100k.fastq.gz
```

An efficient implementation to count the number of lines is also available:

```python
f.count_lines()
```

or reads (assuming 4 lines per read):

```python
f.count_reads()
```
Operators available:

- equality ==

**count_lines ()**
Return number of lines

  This is 10-20 times faster than *wc* on uncompressed file and 3-4 times faster on zipped file (using gunzip -c file | wc -l)

**count_reads ()**
Return count_lines divided by 4

**count_read_gz** *(CHUNKSIZE=65536)*

**extract_head** *(N, output_filename)*
Extract the heads of a FastQ files

  Parameters
  - **N** *(int)* –
  - **output_filename** *(str)* – Based on the extension the output file is zipped or not (.{gz extension only}

  This function is convenient since it takes into account the input file being compressed or not and the output file being compressed ot not. It is in general 2-3 times faster than the equivalent unix commands combined together but is 10 times slower for the case on uncompressed input and uncompressed output.

  **Warning:** this function extract the N first lines and does not check if there are empty lines in your FastQ/FastA files.

**filter** *(identifiers_list=[], min_bp=None, max_bp=None, progressbar=True, output_filename='filtered.fastq', remove=True)*
Filter reads

  Parameters
  - **min_bp** *(int)* – ignore reads with length shorter than min_bp
  - **max_bp** *(int)* – ignore reads with length above max_bp

**joining** *(pattern, output_filename)*

  not implemented

  zcat Block*.fastq.gz | gzip > combined.fastq.gz

**n_lines**
return number of lines (should be 4 times number of reads)

**n_reads**
return number of reads

**next ()**

**random** *(N=10000, output_filename='test.fastq', bp=50, quality=40)*
N here is the number of reads

**rewind ()**
Allows to iter from the beginning without openning the file or creating a new instance.

**select_random_reads** *(N=None, output_filename='random.fastq')*
Select random reads and save in a file
Parameters

- **N (int)** – number of random unique reads to select should provide a number but a list can be used as well. You can select random reads for R1, and re-use the returned list as input for the R2 (since pairs must be kept)

- **output_filename (str)** –

If you have a pair of files, the same reads must be selected in R1 and R2:

```python
f1 = FastQ(file1)
selection = f1.select_random_reads(N=1000)
f2 = FastQ(file2)
f2.select_random_reads(selection)
```

**split_chunks (N=10)**
Not implemented

**split_lines (N=100000, gzip=True)**
Not implemented

**to_kmer_content (k=7)**
Return a Series with kmer count across all reads

Parameters **k (int)** – (default to 7-mers)

**to_krona (k=7, output_filename='fastq.krona')**
Save Krona file with ACGT content within all k-mers

Parameters **k (int)** – (default to 7-mers)

Save results in file, which can then be translated into a HTML file using:

```bash
ktImportText fastq.krona
```

**class FastQC (filename, max_sample=500000, dotile=False, verbose=True)**
Simple QC diagnostic

Similarly to some of the plots of FastQC tools, we scan the FastQ and generates some diagnostic plots. The interest is that we’ll be able to create more advanced plots later on.

Here is an example of the boxplot quality across all bases:

```python
from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.boxplot_quality()
```
Quality scores across all bases

**Warning:** some plots will work for Illumina reads only right now

**Note:** Although all reads are parsed (e.g. to count the number of nucleotides, some information uses a limited number of reads (e.g. qualities), which is set to 500,000 by default.

**constructor**

**Parameters**

- `filename` –
- `max_sample` *(int)* – Large files will not fit in memory. We therefore restrict the numbers of reads to be used for some of the statistics to 500,000. This also reduces the amount of time required to get a good feeling of the data quality. The entire input file is parsed through. This is required for instance to get the number of nucleotides.
- `dotile` *(bool)* – compute more

**boxplot_quality** *(hold=False, ax=None)*

Boxplot quality

Same plots as in FastQC that is average quality for all bases. In addition a 1 sigma error envelope is shown (yellow).
Background separate zone of good, average and bad quality (arbitrary).

\texttt{get_actg_content()}
\texttt{get_stats()}
\texttt{histogram_gc_content()}

Plot histogram of GC content

```python
from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.histogram_gc_content()
```

![GC content distribution (per sequence)](image)

\texttt{histogram_sequence_coordinates()}

Histogram of the sequence coordinates on the plate

```python
from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.histogram_sequence_coordinates()
```

**Note:** in this data set all points have the same coordinates.

\texttt{histogram_sequence_lengths(logy=True)}

Histogram sequence lengths
from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.histogram_sequence_lengths()

imshow_qualities()
Qualities

from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.imshow_qualities()
from pylab import tight_layout; tight_layout()

plot_acgt_content()
Plot histogram of GC content

from sequana import sequana_data
from sequana import FastQC
filename = sequana_data("test.fastq", "testing")
qc = FastQC(filename)
qc.plot_acgt_content()
Kmer module

**build_kmer**(length=6, letters='CG')
Return list of kmer of given length based on a set of letters

*Returns* list of kmers

**get_kmer**(sequence, k=7)
Given a sequence, return consecutive kmers

*Returns* iterator of kmers

IOTools module

**class YamlDocParser**(filename)
A simple parser to extract block content to be found in YAML files

So as to create tooltips automatically in *Sequanix: GUI for snakemake workflows*, one can comment YAML configuration file with block comments (see developers guide in *Developer guide*).

Once read and parsed, all block comments before top-level sections are to be found in the dictionary *sections*.

```python
from sequana import snaketools
from sequana.iotools import YamlDocParser
module = snaketools.Module('quality_control')
r = YamlDocParser(module.config)
r.sections['fastqc']
```
Those lines are removed from the docstring but available as a dictionary

**constructor**

**Parameters** filename (*str*) – the YAML file to parse

```python
# main documentation
# block comment
section1:
    - item
# block comment
section2:
# a comment
section3:
```

Here, section1 and section2 have block comments but not section3

**Taxonomy related (Kraken - Krona)**

```python
class KrakenResults (filename='kraken.out', verbose=True)

Translate Kraken results into a Krona-compatible file

If you run a kraken analysis with KrakenAnalysis, you will end up with a file e.g. named kraken.out (by default).

You could use kraken-translate but then you need extra parsing to convert into a Krona-compatible file. Here, we take the output from kraken and directly transform it to a krona-compatible file.

```python
k = KrakenResults("kraken.out")
k.kraken_to_krona()
```

Then format expected looks like:

|-------|----------------------------------------|----|-----|------|------|------|

Where each row corresponds to one read.

"562:13 561:4 A:31 0:1 562:3" would indicate that:

- the first 13 k-mers mapped to taxonomy ID #562
- the next 4 k-mers mapped to taxonomy ID #561
- the next 31 k-mers contained an ambiguous nucleotide
- the next k-mer was not in the database
- the last 3 k-mers mapped to taxonomy ID #562

See kraken documentation for details.

**Note:** This takes care of fetching taxons and the corresponding lineages from online web services.

---

**2.9. References**
**constructor**

Parameters

- **filename** – the input from KrakenAnalysis class
- **verbose** –

**get_taxonomy_biokit**(*ids*)

Retrieve taxons given a list of taxons

**Parameters**

- **ids** (*list*) – list of taxons as strings or integers. Could also be a single string or a single integer

**Returns**

a dataframe

**Note:** the first call first loads all taxons in memory and takes a few seconds but subsequent calls are much faster

**kraken_to_csv**(*filename, dbname*)

**kraken_to_json**(*filename, dbname*)

**kraken_to_krona**(*output_filename=None, mode=None, nofile=False*)

**Returns**

status: True is everything went fine otherwise False

**plot**(*kind='pie', cmap='copper', threshold=1, radius=0.9, textcolor='red', **kargs*)

A simple non-interactive plot of taxons

**Returns**

None if no taxon were found and a dataframe otherwise

A Krona Javascript output is also available in **kraken_to_krona()**

```python
from sequana import KrakenResults, sequana_data
test_file = sequana_data("test_kraken.out", "testing")
k = KrakenResults(test_file, verbose=False)
df = k.plot(kind='pie')
```
See also:

to generate the data see :class:`KrakenPipeline` or the standalone application :func:`sequana_taxonomy`.

taxons

.. function:: to_js(output='krona.html', onweb=False)

class :class:`KrakenPipeline` (fastq, database, threads=4, output_directory='kraken', verbose=True, dbname=None)

Used by the standalone application :func:`sequana_taxonomy`

This runs Kraken on a set of FastQ files, transform the results in a format compatible for Krona, and creates a Krona HTML report.

```python
from sequana import KrakenTaxon
kt = KrakenPipeline(['R1.fastq.gz', 'R2.fastq.gz'], database='krakendb')
kt.run()
kt.show()
```

**Warning:** We do not provide Kraken database within sequana. You may either download a database from https://ccb.jhu.edu/software/kraken/ or use this class to download a toy example that will be stored in e.g .config/sequana under Unix platforms. See :func:`KrakenDownload`.

See also:
We provide a standalone application of this class, which is called `sequana_taxonomy` and can be used within a command shell.

**Constructor**

**Parameters**

- `fastq` – either a fastq filename or a list of 2 fastq filenames
- `database` – the path to a valid Kraken database
- `threads` – number of threads to be used by Kraken
- `output` – output filename of the Krona HTML page

Description: internally, once Kraken has performed an analysis, reads are associated to a taxon (or not). We then find the corresponding lineage and scientific names to store within a Krona formatted file. `KtImportTex` is then used to create the Krona page.

```python
def run()
    Run the analysis using Kraken and create the Krona output
```

```python
def show()
    Opens the filename defined in the constructor
```

class `KrakenAnalysis` *(fastq, database, threads=4)*

Run kraken on a set of FastQ files

In order to run a Kraken analysis, we first need a local database. We provide a Toy example. The ToyDB is downloadable as follows (you will need to run the following code only once):

```python
from sequana import KrakenDownload
kd = KrakenDownload()
kd.download_kraken_toydb()
```

See also:

*KrakenDownload* for more database and *sequana.kraken_builder.KrakenBuilder* to build your own databases

The path to the database is required to run the analysis. It has been stored in the directory `./config/sequana/kraken_toydb` under Linux platforms. The following code should be platform independent:

```python
import os
from sequana import sequana_config_path
database = sequana_config_path + os.sep + "kraken_toydb"
```

Finally, we can run the analysis on the toy data set:

```python
from sequana import sequana_data
data = sequana_data("Hm2_GTGAAA_L005_R1_001.fastq.gz", "data")
ka = KrakenAnalysis(data, database=database)
ka.run()
```

This creates a file named `kraken.out`. It can be interpreted with `KrakenResults`
**Constructor**

**Parameters**

- **fastq** – either a fastq filename or a list of 2 fastq filenames
- **database** – the path to a valid Kraken database
- **threads** – number of threads to be used by Kraken
- **output** – output filename of the Krona HTML page
- **return** –

**run** *(output_filename=None)*

Performs the kraken analysis

**Parameters**

- **output_filename** *(str)* – if not provided, a temporary file is used and stored in kraken_output.

**class** **KrakenDownload**

Utility to download Kraken DB and place them in a local directory

```python
from sequana import KrakenDownload
kd = KrakenDownload()
kd.download('toydb')
kd.download('minikraken')
```

A large database (8Gb) is available on synapse and has the following DOI:

```
doi:10.7303/syn6171000
```

It can be downloaded manually or if you have a Synapse login (https://www.synapse.org), you can use:

```python
from sequana import KrakenDownload
kd = KrakenDownload()
kd.downloaded("sequana_db1")
```

**download** *(name, verbose=True)*

dv = <easydev.tools.DevTools object>

**class** **KronaMerger** *(filename)*

Utility to merge two Krona files

Imagine those two files (formatted for Krona; first column is a counter):

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14011</td>
<td>Bacteria</td>
<td>Proteobacteria</td>
<td>species1</td>
</tr>
<tr>
<td>591</td>
<td>Bacteria</td>
<td>Proteobacteria</td>
<td>species4</td>
</tr>
<tr>
<td>184</td>
<td>Bacteria</td>
<td>Proteobacteria</td>
<td>species3</td>
</tr>
<tr>
<td>132</td>
<td>Bacteria</td>
<td>Proteobacteria</td>
<td>species2</td>
</tr>
<tr>
<td>32</td>
<td>Bacteria</td>
<td>Proteobacteria</td>
<td>species1</td>
</tr>
</tbody>
</table>

You can merge the two files. The first and last lines correspond to the same taxon (species1) so we should end up with a new Krona file with 4 lines only.

The test files are available within Sequana as test_krona_k1.tsv and test_krona_k2.tsv:

2.9. References
from sequana import KronaMerger, sequana_data
k1 = KronaMerger(sequana_data("test_krona_k1.tsv"))
k2 = KronaMerger(sequana_data("test_krona_k2.tsv"))
k1 += k2
# Save the results. Note that it must be tabulated for Krona external usage
k1.to_tsv("new.tsv")

Warning: separator must be tabulars

**constructor**

Parameters **filename**(str) –

to_tsv(output_filename)
Save the content into a new file in TSV format

class **KrakenBuilder**(dbname)
This class will help you building a custom Kraken database

You will need a few steps, and depending on the FASTA files you want to include lots of resources (memory and space wise). In the following example, we will be reasonable and use only viruses FASTA files.

First, we need to create the data structure directory. Let us call it **virusdb**:

```python
from sequana import KrakenBuilder
kb = KrakenBuilder("virusdb")
```

We then need to download a large taxonomic database from NCBI. You may already have a local copy, in which case you would need to copy it in virusdb/taxonomy directory. If not, type:

```python
kb.download_taxonomy()
```

The virusdb/taxonomy directory will contain about 8.5G of data.

Note that this currently requires the unix tools **wget** and **tar**.

Then, we need to add some fasta files. You may download specific FASTA files if you know the accession numbers using `download_accession()`. However, we also provide a method to download all viruses from ENA:

```python
kb.download_viruses()
```

This will take a while to download the more than 4500 FASTA files (10 minutes on a good connection). You will end up with a data set of about 100 Mb of FASTA files.

If you wish to download other FASTA (e.g. all bacteria), you will need to use another class from the **sequana. databases**:

```python
from sequana.databases import ENADownload
ena = ENADownload()
ena.download_fasta("bacteria.txt", output_dir="virusdb/library/added")
```

Please see the documentation for more options and list of species to download.

It is now time to build the DB itself. This is based on the kraken tool. You may do it yourself in a shell:
kraken-build --rebuild -db virusdb --minimizer-len 10 --max-db-size 4 --threads 4
--kmer-len 26 --jellyfish-hash-size 500000000

Or you the KrakenBuilder. First you need to look at the params attribute. The most important key/value that affect the size of the DB are:

- **kb.params['kmer_length']** (max value is 31)
- **kb.params['max_db_size']** is the max size of the DB files in Gb
- **kb.params['minimizer_len']**

To create a small DB quickly, we set those values:

- **kb.params['kmer_length']** = 26
- **kb.params['minimizer_len']** = 10

However, for production, we would recommend 31 and 13 (default)

This takes about 2 minutes to build and the final DB is about 800Mb.

Lots of useless files are in the directory and can be removed using kraken itself. However we do a little bit more and therefore have our own cleaning function:

- **kb.clean_db()**

Kraken-build uses jellyfish. The **hash_size** parameter is the jellyfish hash_size parameter. If you set it to 6400M, the memory required is about 6.9 bytes times 6400M that is 40Gb of memory. The default value used here means 3.5Gb are required.

The size to store the DB itself should be

\[
\text{Math } sD + 8 (4^M)
\]

where \(s\) is about 12 bytes (used to store a kmer/taxon pair, \(D\) is the number of kmer in the final database, which cannot be estimated beforehand, and \(M\) the length minimiser parameter.

**Constructor**

- **Parameters dbname** (str) – Create the Kraken DB in this directory

- **clean_db()**
  
  Once called, you will not be able to append more FASTA files

- **download_accession**(acc)

  Download a specific Fasta from ENA given its accession number

  Note that if you want to add specific FASTA from ENA, you must use that function to make sure the header will be understood by Kraken; The header must use a GI number (not ENA)

- **download taxonomy** (force=False)

  Download kraken data

  The downloaded file is large (1.3Gb) and the unzipped file is about 9Gb.

  If already present, do not download the file except if the force parameter is set to True.

- **download viruses**

- **get_gis**(extensions=['fa'])

- **get taxons from gis** (gis, filename='gi_taxid_nucl.dmp')
init()

run(dbs=[], download_taxon=True)
Create the Custom Kraken DB
1. download taxonomy files
2. Load the DBs (e.g. viruses)
3. Build DB with kraken-build
4. Clean it up

Pacbio module

class BAMPacbio(filename)
BAM reader for Pacbio (reads)

You can read a file as follows:

```python
from sequana.pacbio import BAMPacbio
from sequana import sequana_data
filename = sequana_data("test_pacbio_subreads.bam")
b = BAMPacbio(filename)
```

A summary of the data is stored in the attribute `df`. It contains information such as the length of the reads, the ACGT content, the GC content.

Several plotting methods are available. For instance, `hist_snr()`.

Constructor

Parameters

**filename (str)** – filename of the input pacbio BAM file. The content of the BAM file is not the output of a mapper. Instead, it is the output of a Pacbio (Sequel) sequencing (e.g., subreads).

**df**

filter_length(output_filename, threshold_min=0, threshold_max=inf)
Write reads within the length range to BAM output

Parameters

- **output_filename (str)** – name of output file
- **threshold_min (int)** – minimum length
- **threshold_max (int)** – maximum length

hist_GC(bins=50, alpha=0.5, hold=False, fontsize=12, grid=True, xlabel='GC %', ylabel='#', label='')
Plot histogram GC content

Parameters

- **bins (int)** – binning for the histogram
- **alpha (float)** – transparency of the histograms
- **hold (bool)** –
- **fontsize (int)** –
from sequana.pacbio import BAMPacbio
from sequana import sequana_data
b = BAMPacbio(sequana_data("test_pacbio_subreads.bam"))
b.hist_GC()

hist_ZMW_subreads (alpha=0.5, hold=False, fontsize=12, grid=True, xlabel='Number of ZMW passes', ylabel='#', label='')
Plot histogram of number of reads per ZMW (number of passes)

Parameters
• alpha (float) – transparency of the histograms
• hold (bool) –
• fontsize (int) –
• grid (bool) –
• xlabel (str) –
• ylabel (str) –
• label (str) –
```python
from sequana.pacbio import BAMPacbio
from sequana import sequana_data
b = BAMPacbio(sequana_data("test_pacbio_subreads.bam"))
b.hist_ZMW_subreads()
```

**Number of ZMW passes**

![Number of ZMW passes](image)

```python
b.hist_len(bins=50, alpha=0.5, hold=False, fontsize=12, grid=True, xlabel='Read Length', ylabel='#', label='')
```

Plot histogram Read length

**Parameters**

- `bins (int)` – binning for the histogram
- `alpha (float)` – transparency of the histograms
- `hold (bool)` –
- `fontsize (int)` –
- `grid (bool)` –
- `xlabel (str)` –
- `ylabel (str)` –
- `label (str)` –

```python
from sequana.pacbio import BAMPacbio
from sequana import sequana_data
b = BAMPacbio(sequana_data("test_pacbio_subreads.bam"))
b.hist_len()
```
hist_snr (bins=50, alpha=0.5, hold=False, fontsize=12, grid=True, xlabel='SNR', ylabel='#')

Plot histogram of the ACGT SNRs for all reads

Parameters

- bins (int) – binning for the histogram
- alpha (float) – transparency of the histograms
- hold (bool) –
- fontsize (int) –
- grid (bool) –
- xlabel (str) –
- ylabel (str) –

```python
from sequana.pacbio import BAMPaclbio
from sequana import sequana_data
b = BAMPaclbio(sequana_data("test_pacbio_subreads.bam"))
b.hist_snr()
```
```python
from sequana.pacbio import BAMPacbio
from sequana import sequana_data
b = BAMPacbio(sequana_data("test_pacbio_subreads.bam"))
b.plot_GC_read_len()
```

**nb_pass**

**plot_GC_read_len**(alpha=0.07, hold=False, fontsize=12, grid=True, xlabel='GC %', ylabel='#')

Plot GC content versus read length

**Parameters**

- **alpha**(float) — transparency of the 2D histogram
- **hold**(bool) —
- **fontsize**(int) —
- **grid**(bool) —
- **xlabel**(str) —
- **ylabel**(str) —
reset()

**stride** *(output_filename, stride=10, shift=0, random=False)*

Write a subset of reads to BAM output

**Parameters**

- **output_filename** *(str)* – name of output file
- **stride** *(int)* – optional, number of reads to read to output one read
- **shift** *(int)* – number of reads to ignore at the beginning of input file
- **random** *(bool)* – if True, at each step the read to output is randomly selected

## Phred quality

Manipulate phred quality of reads

FastQ quality are stored as characters. The phred scales indicates the range of characters.

In general, characters goes from ! to ~ that is from 33 to 126 in an ascii table. This convention starts at 33 because characters before ! may cause trouble (e.g. white spaces). This scale is the Sanger scale. There are 2 other scales that could be used ranging from 59 to 126 (illumina 1) and from 64 to 126 (illumina 1.3+).

So, here are the offset to use:

<table>
<thead>
<tr>
<th>Name</th>
<th>offset</th>
<th>Numeric range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sanger</td>
<td>33</td>
<td>0 to 93</td>
</tr>
<tr>
<td>Solexa</td>
<td>64</td>
<td>-5 to 62</td>
</tr>
<tr>
<td>illumina1.3+</td>
<td>64</td>
<td>0 to 62</td>
</tr>
</tbody>
</table>
Even though dedicated tools would have better performances, we provide a set of convenient functions. An example is provided here below to plot the quality corresponding to a character string extracted from a FastQ read.

In this example, we use `Quality` class where the default offset is 33 (Sanger). We compare the quality for another offset.

```python
from sequana import phred

from sequana.phred import Quality
q = Quality('BCCFFFFFHHHHHIJJJJJJIIJJJJJJJJFH')
q.plot()
q.offset = 64
q.plot()
from pylab import legend
legend(loc="best")
```

```python
class Quality(seq, offset=33)
Phred quality

>>> from sequana.phred import Quality
>>> q = Quality('BCCFFFFFHHHHHIJJJJJJIIJJJJJJJJFH')
>>> q.plot()
```
You can access to the quality as a list using the `quality` attribute and the mean quality from the `mean_quality` attribute.

```python
mean_quality
    return mean quality
```

```python
plot (fontsize=16)
    plot quality versus base position
```

```python
quality
    phred string into quality list
```

## Running median

Data analysis tool

```python
RunningMedian(data, width[, container])

Running median (fast)
```

```python
class RunningMedian (data, width, container=<class 'list'>)

Running median (fast)
```

This is an efficient implementation of running median, faster than SciPy implementation v0.17 and a skip list method.

The main idea comes from a recipe posted in this website: http://code.activestate.com/recipes/576930/#c3 that uses a simple list as proposed in https://gist.github.com/f0k/2f8402e4dfb6974bf6c1 and was adapted to our needs
included object oriented implementation.

**Note:** a circular running median is implemented in `sequana.bedtools.GenomeCov`

```python
from sequana.running_median import RunningMedian
rm = RunningMedian(data, 101)
results = rm.run()
```

**Warning:** the first W/2 and last W/2 positions should be ignored since they do not use W values. In this implementation, the last W/2 values are currently set to zero.

This shows how the results agree with scipy

```python
from pylab import *
import scipy.signal
from sequana.running_median import RunningMedian

cf()
x = randn(100)
plot(x, 'k')
plot(RunningMedian(x, 9).run(), 'r', lw=4)
plot(scipy.signal.medfilt(x, 9), 'go')
grid()
```

![Graph showing the results agree with scipy](image-url)
```python
from sequana.running_median import RunningMedian
from pylab import *

N = 1000
X = linspace(0, N-1, N)

# Create some interesting data with SNP and longer over represented section.
data = 20 + randn(N) + sin(X*2*pi/1000.*5)
data[300:350] += 10
data[500:505] += 100
data[700] = 1000

plot(X, data, "k", label="data")
rm = RunningMedian(data, 101)
plot(X, rm.run(), 'r', label="median W=201")

from sequana.stats import moving_average as ma
plot(X[100:-100], ma(data, 201), 'g', label="mean W=201")
grid()
legend()
ylim([10, 50])
```

Note that for visualisation, we set the ylimits to 50 but the data at position 500 goes up to 120 and there is an large outlier (1000) at position 700.

We see that the median is less sensible to the outliers, as expected. The median is highly interesting for large

2.9. References
outliers on short duration (e.g. here the peak at position 500) but is also less biased by larger regions.

**Note:** The beginning and end of the running median are irrelevant. There are actually equal to the data in our implementation.

**Note:** using blist instead of list is not always faster. It depends on the width of the window being used. list and blist are equivalent for W below 20,000 (list is slightly faster). However, for large W, blist has an $O(\log(n))$ complexity while list has a $O(n)$ complexity.

**constructor**

**Parameters**

- **data** – your data vector
- **width** – running window length
- **container** – a container (defaults to list). Could be a B-tree blist from the blist package but is 30% slower than a pure list for $W < 20,000$

scipy in $O(n)$ list in $\sqrt{n}$ blist in $O(\log(n))$

```python
run()
```

```python
running_median(data, width, container=<class 'list'>)
```

**Snakemake module**

Set of tools to manipulate Snakefile and config files

Here is an overview (see details here below)

<table>
<thead>
<tr>
<th>sequana.snaketools.DOTPParser</th>
<th>Utility to manipulate the dot file returned by Snakemake</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequana.snaketools.FastQFactory</td>
<td>FastQ Factory tool</td>
</tr>
<tr>
<td>sequana.snaketools.FileFactory</td>
<td>Factory to handle a set of files</td>
</tr>
<tr>
<td>sequana.snaketools.Module</td>
<td>Data structure that holds metadata about a Module</td>
</tr>
<tr>
<td>sequana.snaketools.ModuleFinderSingleton</td>
<td>Data structure to hold the Module names</td>
</tr>
<tr>
<td>sequana.snaketools.PipelineManager</td>
<td>Utility to manage easily the snakemake pipeline</td>
</tr>
<tr>
<td>sequana.snaketools.SequanaConfig</td>
<td>Reads YAML config file and ease access to its contents</td>
</tr>
<tr>
<td>sequana.snaketools.message</td>
<td>Dedicated print function to include in Snakefiles</td>
</tr>
<tr>
<td>sequana.snaketools.modules</td>
<td>dictionary with module names as keys and fullpath to the Snakefile as values</td>
</tr>
</tbody>
</table>

**class DOTParser (filename)**

Utility to manipulate the dot file returned by Snakemake

This class is used in the `dag` and `rulegraph` rules used in the snakemake pipeline. The input must be a dag/rulegraph created by snakemake.

Consider this example where the test file was created by snakemake `--dag`
from sequana import sequana_data
from sequana.snaketools import DOTParser

filename = sequana_data("test_dag.dot")
dot = DOTParser(filename)

# creates test_dag.ann.dot locally
dot.add_urls("test.dot", {"fastqc": "fastqc.html"})

You can then convert the dag in an unix shell:

dot -Tsvg test.ann.dot -o test.svg

**constructor**

Parameters **filename** *(str)* – a DAG in dot format created by snakemake

**add_urls** *(output_filename=None, mapper={})*

Change the dot file adding URL on some nodes

Parameters

- **output_filename** *(str)* – the DAG file in dot format (graphviz)
- **mapper** *(dict)* – a dictionary where keys are named after the rule names for which an HTML will be available (provided here as keys)
class **FastQFactory** *(pattern, extension=['fq.gz', 'fastq.gz'], strict=True, verbose=False)*

FastQ Factory tool

In NGS experiments, reads are stored in a so-called FastQ file. The file is named:

 PREFIX_R1_SUFFIX.fastq.gz

where _R1_ tag is always to be found. This is a single-ended case. In paired case, a second file is to be found:

 PREFIX_R2_SUFFIX.fastq.gz

The PREFIX indicates the sample name. The SUFFIX does not convey any information per se.

Yet, in long reads experiments (for instance), naming convention is different and may nor be single/paired end convention.

In a directory (recursively or not), there could be lots of samples. This class can be used to get all the sample prefix in the *tags* attribute.

Given a tag, one can get the corresponding file(s):

```python
ff = FastQFactory("*fastq.gz")
ff.tags
ff.get_file1(ff.tags[0])
len(ff)
```

**Constructor**

**Parameters**

• *pattern*(str) – a global pattern (e.g., H*fastq.gz)

• *extension*(list) – not used

• *strict*(bool) – if true, the pattern _R1_ or _R2_ must be found

• *strict–*

• *verbose*(bool)–

**get_file1**(tag=None)

**get_file2**(tag=None)

class **FileFactory** *(pattern)*

Factory to handle a set of files

```python
from sequana.snaketools import FileFactory
ff = FileFactory("H*.gz")
ff.filenames
```

A set of useful methods are available based on this convention:

```python
>>> fullpath = /home/user/test/A.fastq.gz
>>> dirname(fullpath)
"/home/user/test"
>>> basename(fullpath)
'A.fastq.gz'
>>> realpath(fullpath) # is .., expanded to /home/user/test
```
>>> all_extensions
"fastq.gz"
>>> extensions
".gz"

**Constructor**

Parameters `pattern` – can be a filename, list of filenames, or a global pattern (a unix regular expression with wildcards). For instance, “*/fastq.gz”

**Warning:** Only in Python 3.X supports the recursive global pattern for now.

all_extensions
the extensions (list)

basenames
list of filenames and their extensions without the path

extensions
the last extension (list)

filenames
list of filenames (no path, no extension)

pathname
the common relative path

pathnames
the relative path for each file (list)

realpaths
real path is the full path + the filename the extension

class Module *(name)*
Data structure that holds metadata about a Module

In Sequana, we provide rules and pipelines to be used with snakemake. Snakemake rules look like:

```python
rule <name>:
    :input: file1
    :output: file2
    :shell: "cp file1 file2"
```

A pipeline may look like:

```python
include: "path_to_rule1"
include: "path_to_rule2"
rule all:
    input: FINAL_FILES
```

Note that the pipeline includes rules by providing the path to them.

All rules can be stored in a single directory. Similarly for pipelines. We decided not to use that convention. Instead, we bundle rules (and pipelines) in their own directories so that other files can be stored with them. We also consider that

2.9. References
1. If the Snakefile includes other Snakefile then it is Pipeline.

2. Otherwise it is a simple Rule.

So, a Module in sequana’s parlance is a directory that contains a rule or a pipeline and associated files. There is currently no strict conventions for rule Modules except for their own rule file. However, pipeline Modules should have the following files:

- A snakemake file named after the directory with the extension .rules
- A README.rst file in restructured text format
- An optional config file in YAML format named config.yaml. Although json format is possible, we use YAML throughout sequana for consistency. If not found, the config.yaml is taken from the parent directory. Rules do not have any but pipelines do. So if a pipeline does not provide a config.yaml, the one found in ./sequana/sequana/pipelines will be used.
- A requirements.txt

**Note:** Developers who wish to include new rules should refer to the Developer guide.

**Note:** It is important that module’s name should be used to name the directory and the rule/pipeline.

The Modules are stored in sequana/rules and sequana/pipelines directories. The modules’ names cannot be duplicated.

Example:

```
pipelines/test_pipe/test_pipe.rules
pipelines/test_pipe/README.rst
rules/rule1/rule1.rules
rules/rule1/README.rst
```

The Module will ease the retrieval of information linked to a rule or pipeline. For instance if a pipeline has a config file, its path can be retrieved easily:

```
m = Module("quality_control")
m.config
```

This Module may be rule or pipeline, the method `is_pipeline()` can be used to get that information. Other useful methods are available such as `onweb()` that open the web page of the pipeline (linked to the README).

**Constructor**

- **Parameters** name (str) – the name of an available module.
- **check** (mode='warning')
- **cluster_config**
  - full path to the config cluster file of the module
- **config**
  - full path to the config file of the module
- **config_cluster**
  - full path to the config cluster file of the module
**description**
Content of the README file associated with

**is_executable** *(verbose=False)*
Is the module executable
A Pipeline Module should have a requirements.txt file that is introspected to check if all executables are available;

**Parameters verbose** –

**Returns** a tuple. First element is a boolean to tell if it executable. Second element is the list of missing executables.

**is_pipeline()**
Return true is this module is a pipeline

**name**
name of the module

**onweb()**

**overview**

**path**
full path to the module directory

**readme**
full path to the README file of the module

**requirements**
list of requirements

**snakefile**
full path to the Snakefile file of the module

**class PipelineManager** *(name, config, pattern='*.fastq.gz')*
Utility to manage easily the snakemake pipeline

Inside a snakefile, use it as follows:

```python
from sequana import PipelineManager
manager = PipelineManager("pipeline_name", "config.yaml")
```

config file must have these fields:

```yaml
- input_directory: #a_path
- input_extension: fastq.gz  # this is the default. could be also fq.gz
- input_pattern: # a_global_pattern e.g. H*fastq.gz
- input_samples:
  - file1:
  - file2:
```

The manager can then easily access to the data with a `FastQFactory` instance:

```python
manager.ff.filenames
```

This can be further used to get a wildcards with the proper directory.

The manager also tells you if the samples are paired or not assuming all samples are homogneous (either all paired or all single-ended).
If there is only one sample, the attribute `mode` is set to “nowc” meaning no wildcard. Otherwise, we assume that we are in a wildcard mode.

When the mode is set, two attributes are also set: `sample` and `basename`.

If the mode os `nowc`, the `sample` and `basename` are hardcoded to the sample name and sample/rule/sample respectively. Whereas in the `wc` mode, the sample and basename are wildcard set to “{sample}” and “{sample}/rulename/{sample}”. See the following methods :meth:``

For developers: the config attribute should use for getter only

**Constructor**

**Parameters**

- `name` – name of the pipeline
- `config` – name of a configuration file
- `pattern` – a default pattern if not provided in the configuration file as an `input_pattern` field.

**error** *(msg)*

**getlogdir** *(rulename)*

Create log directory: */sample/logs/sample_rule.logs

**getname** *(rulename, suffix=None)*

Returns basename % rulename + suffix

**getrawdata** *

Return list of raw data

If mode is nowc, a list of files is returned (one or two files) otherwise, a function compatible with snakemake is returned. This function contains a wildcard to each of the samples found by the manager.

**getreportdir** *(acronym)*

Create the report directory.

**gettokdir** *(rulename)*

**class** `SnakeMakeStats` *(filename, N=1)*

Interpret the snakemake stats file

Run the Snakemake with this option:

```
-- stats stats.txt
```

Then:

```python
from sequana.snakertools import SnakeMakeStats
from sequana import sequana_data
filename = sequana_data("test_snakemake_stats.txt", "testing")
s = SnakeMakeStats(filename)
s.plot()
```
Constructor

```
plot (fontsize=16)
    Create the barplot from the stats file

plot_and_save (filename='snakemake_stats.png', outputdir='report')
```

```
class SequanaConfig (data=None, converts_none_to_str=True)
    Reads YAML config file and ease access to its contents

    This can also be used to check the validity of the config file

    >>> sc = SequanaConfig(config)
    >>> sc.config.pattern == "*.fastq.gz"
    True
```

Input files should be stored into:

```
input_samples:
    - file1: FILE1
    - file2: FILE2
```

The second file may be optional.

Empty strings in a config are interpreted as None but SequanaConfig will replace None with empty strings, which is probably what was expected from the user. Similarly, in snakemake when settings the config file, one can override a value with a False but this is intercepted as “False” This will transform back the “True” into True.
Could be a JSON or a YAML file

Parameters **filename** (*str*) – filename to a config file in json or YAML format.

SEQUANA config files must have some specific fields:

```python
input_directory
input_samples...
```

```python
check_sequana_fields()
cleanup()
    # Remove template elements and change None to empty string.
cleanup_config()
copy_requirements(target)
    # Copy files to run the pipeline
    # If a requirement file exists, it is copied in the target directory. If not, it can be either an http resources or a sequana resources.
save (filename=’config.yaml’, cleanup=True)
    # Save the yaml code in _yaml_code with comments
```

```python
modules = {‘dsrc_to_gz’: ‘/home/docs/checkouts/readthedocs.org/user_builds/sequana/conda/develop/lib/python3.5/site-packages/sequana-0.1.21-py3.5.egg/sequana/rules/Adapters/adapter_removal/adapter_removal.rules’}
```

### Snpeff module

Tools to launch snpEff.

**class Snpeff (reference, file_format=’, log=None)**

Python wrapper to set and launch snpEff from a genbank file. It is not easy to use a custom genbank file with snpEff.

Example:

```python
snpeff = Snpeff(‘file.gbk’)
snpeff.launch_snpeff(‘variants.vcf’, ‘variant.ann.vcf’)
```

### Constructor

**Parameters**

- **reference** – annotation reference.
- **file_format** – format of your file. (‘only genbank actually’)
- **log** – log file

**add_locus_in_fasta (fasta, output_file)**

Add locus of annotation file in description line of fasta file.

**Parameters**

- **fasta** (*str*) – input fasta file where you want to add locus.
- **output_file** (*str*) – output file.

**extension** = {‘gff’: ‘.gff’, ‘genbank’: ‘.gbk’, ‘gtf’: ‘.gtf’}
**launch_snpeff** *(vcf_filename, output, html_output=None, options='')*

Launch snpEff with the custom genbank file.

**Parameters**
- **vcf_filename** *(str)* – input VCF filename.
- **output** *(str)* – output VCF filename.
- **html_output** *(str)* – filename of the HTML creates by snpEff.
- **options** *(str)* – any options recognised by snpEff.

**downloadfasta_and_genbank** *(identifier, tag, genbank=True, fasta=True)*

**Parameters**
- **identifier** – valid identifier to retrieve from NCBI (genbank) and ENA (fasta)
- **tag** – name of the filename for the genbank and fasta files.

**General tools**

**misc utilities**

**textwrap** *(text, width=80, indent=0)*

Wrap a string with 80 characters

**Parameters**
- **text** – input text
- **width** – (defaults to 80 characters)
- **indent** – possible indentation (0 by default)

**rest2html** *(s)*

Converts a restructuredText document into HTML

Note that the returned object is a bytes so need to be decoded with decode()

**wget** *(link, output)*

Retrieve a file from internet.

**Parameters**
- **link** *(str)* – a valid URL
- **output** *(str)* – the output filename

**Warning:** no sanity check of any kind for now

**Todo**

move to easydev

**findpos** *(seq, chr)*

Find position(s) of a substring into a longer string.

Note that this function is a generator:

2.9. References
```python
>>> list(findpos("AACCAGAAGT", "GG"))
[4,8]
```

**on_cluster** *(pattern=’tars-‘)*

Used to check if we are on a cluster

“tars-” is the name of a cluster’s hostname. Change or append the argument **pattern** with your cluster’s hostname

**Parameters** **pattern** *(str)* – a list of names (strings) or a string

**Statistical tools**

**moving_average** *(data, n)*

Compute moving average

**Parameters** **n** – window’s size (odd or even).

```python
>>> from sequana.stats import moving_average as ma
>>> ma([[1,1,1,1,3,3,3,3], 4])
array([ 1. , 1.5, 2. , 2.5, 3. ])
```

**Note:** the final vector does not have the same size as the input vector.

**evenness** *(data)*

Return Evenness of the coverage

**Reference** Konrad Oexle, Journal of Human Genetics 2016, Evaluation of the evenness score in NGS.

work before or after normalisation but lead to different results.

**General tools**

**class** **StatsBAM2Mapped** *(dict(mapping) -> new dictionary initialized from a mapping object’s (key, value) pairs dict(Iterable) -> new dictionary initialized as if via: d = { } for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)*

**to_html** *(with_stats=True)*

**bam_to_mapped_unmapped_fastq** *(filename, output_directory=None, verbose=True)*

Create mapped and unmapped fastq files from a BAM file

**Context** given a reference, one or two FastQ files are mapped onto the reference to generate a BAM file. This BAM file is a compressed version of a SAM file, which interpretation should be eased within this function.

**Parameters**

- **filename** – input BAM file
- **output_directory** – where to save the mapped and unmapped files

**Returns** dictionary with number of reads for each file (mapped/unmapped for R1/R2) as well as the mode (paired or not), the number of unpaired reads, and the number of duplicated reads. The unpaired reads should be zero (sanity check)

Given a BAM file, create FASTQ with R1/R2 reads mapped and unmapped. In the paired-end case, 4 files are created.
Note that this function is efficient in that it does not create intermediate files limiting IO in the process.

**Details** Secondary alignment (flag 256) are dropped so as to remove any ambiguous alignments. The output dictionary stores “secondary” key to keep track of the total number of secondary reads that are dropped. If the flag is 256 and the read is unpaired, the key *unpaired* is also incremented.

If the flag is not equal to 256, we first reverse complement reads that are tagged as *reverse* in the BAM file. Then, reads that are not paired or not “proper pair” (neither flag 4 nor flag 8) are ignored.

If R1 is mapped or R2 is mapped then the reads are considered mapped. If both R1 and R2 are unmapped, then reads are unmapped.

---

**Note:** about chimeric alignment: one is the representative and the other is the supplementary. This flag is not used in this function. Note also that chimeric alignment have same QNAME and flag 4 and 8

---

**Todo**

comments are missing since there are not stored in the BAM file.

---

**VCF module**

Analysis of VCF file generated by freebayes.

**class** `Filtered_freebayes` *(variants, fb_vcf)*

Variants filtered with VCF_freebayes.

**constructor**

**Parameters**

- `variants` *(list)* – list of variants record.
- `bcf` *(BCF_freebayes)* – class parent.

**df**

**to_csv** *(output_filename)*

Write DataFrame in CSV format.

**Params** `str` `output_filename` output CSV filename.

**to_vcf** *(output_filename)*

Write BCF file in VCF format.

**Params** `str` `output_filename` output VCF filename.

**variants**

Get the variant list.

**class** `VCF_freebayes` *(filename, **kwargs)*

VCF class (Variant Calling Format)

This class is a wrapping of vcf.Reader class from the pyVCF package. It is dedicated for VCF file generated by freebayes. A data frame with all variants is produced which can be write as csv file. It can filter variants with a dictionary of filter parameter. Filter variants are wrote in a new VCF file.
```python
from sequana import sequana_data
from sequana.freebayes_vcf_filter import VCF_freebayes
vcf_filename = sequana_data("JB409847.vcf")

# Read the data
v = VCF_freebayes(vcf_filename)

# Filter the data
filter_dict = {
    "freebayes_score": 200,
    "frequency": 0.8,
    "min_depth": 10,
    "forward_depth": 3,
    "reverse_depth": 3,
    "strand_ratio": 0.2
}
filter_v = v.filter_vcf(filter_dict)
filter_v.to_vcf('output.filter.vcf')
```

**constructor**

Parameters

- **filename** *(str)* – a vcf file.
- **kwargs** – any arguments accepted by vcf.Reader

filter_vcf *(filter_dict=None)*

Filter variants in the VCF file.

Parameters **filter_dict** *(dict)* – dictionary of filters. It updates the attribute VCF_freebayes.filters

Return Filtered_freebayes object.

filters_params

Get or set the filters parameters to select variants of interest. Setter take a dictionary as parameter to update the attribute VCF_freebayes.filters_params. Delete will reset different variable to 0.

```python
v = VCF_freebayes("input.bcf")
v.filter_params = {
    "freebayes_score": 200,
    "frequency": 0.8,
    "min_depth": 10,
    "forward_depth": 3,
    "reverse_depth": 3,
    "strand_ratio": 0.2
}
```

**rewind()**

Rewind the reader

class Variant *(record)*

Variant class to stock variant reader and dictionary that resume most important informations.

**constructor**

Parameters

- **record** *(RecordVariant)* – variant record
• **resume** *(dict)* – most important informations of variant

```python
record
resume
```

**compute_frequency**(vcf_line)

Compute frequency of alternate allele. \( \text{alt}_\text{freq} = \frac{\text{Count Alternate Allele}}{\text{Depth}} \)

**Parameters**

- **vcf_line** *(vcf.model._Record)* – variant record

**compute_strand_balance**(vcf_line)

Compute strand balance of alternate allele include in \([0,0.5]\). \( \text{strand}_\text{bal} = \frac{\text{Alt Forward}}{\text{(Alt Forward + Alt Reverse)}} \)

**Parameters**

- **vcf_line** *(vcf.model._Record)* – variant record

**strand_ratio**(number1, number2)

Compute ratio between two number. Return result between \([0:0.5]\).

**Reports**

Data structure to build reports in a consistent way  Report dedicated to BAM file

```python
BAMReport(**kargs)**
```

**Others**

Retrieve data from sequana library

**sequana_data** *(filename=None, where=None)*

Return full path of a sequana resource data file.

**Parameters**

- **filename** *(str)* – a valid filename to be found
- **where** *(str)* – one of the registered data directory (see below)

**Returns** the path of file. See also here below in the case where filename is set to “*”.

```python
from sequana import sequana_data
filename = sequana_data("test.bam")
```

Type the function name with “*” parameter to get a list of available files. With the where argument set, the function returns a list of files. Without the where argument, a dictionary is returned where keys correspond to the registered directories:

```python
filenames = sequana_data("*", where="images")
```

Registered directories are:

- *data*
- *testing*
- *data/adapters*
• images

Note: this does not handle wildcards. The * means retrieve all files.

Some useful data sets to be used in the analysis

The command `sequana.sequana_data()` may be used to retrieved data from this package. For example, a small but standard reference (phix) is used in some NGS experiments. The file is small enough that it is provided within sequana and its filename (full path) can be retrieved as follows:

```python
from sequana import sequana_data
fullpath = sequana_data("phiX174.fa", "data")
```

Other files stored in this directory will be documented here.

**FAQS**

**Conda related**

Create a conda environment on IP cluster:

```bash
module load conda
conda create --name py35 python=3.5
source condaenvs/py35/bin/activate py35
```

add channel from where to download packages:

```bash
conda config --add channels r bioconda
```

**What are the dependencies**

There are two kind of dependencies. First, the Python libraries such as matplotlib or Pandas. Second, the external tools such as BWA (alignment) or Kraken (taxonomy). The first kind of tools can be installed using Anaconda and the default conda channel. For instance:

```bash
conda install pandas
```

The second kind of tools can also be installed using another conda channel called `bioconda`. For instance:

```bash
conda install bwa
```

The full list of dependencies will be maintained in the installation section.

**Installation issues**

As explained in the previous section, most of the dependencies can be installed via Conda. If not, pip is recommended. Yet there are still a few dependencies that needs manual installation.
quast

http://quast.bioinf.spbau.ru/manual.html#sec1

```
wget https://downloads.sourceforge.net/project/quast/quast-4.2.tar.gz
tar -xzf quast-4.2.tar.gz
cd quast-4.2
```

Alternatively, get the source code from their GitHub (takes a while):

```
git clone https://github.com/ablab/quast
cd quast
python setup.py install
```

graphviz

graphviz provides an executable called dot. If you type dot in a shell and get this error message:

```
Warning: Could not load ...
lib/graphviz/libgvplugin_gd.so.6" - file not found
```

This may be solved by re-installation graphviz using the main anaconda channel (instead of bioconda):

```
conda install --override-channels -c anaconda graphviz=2.38.0
```

**Update April 2017** replace anaconda with conda-forge

matplotlib

If you get errors related to the X connection, you may need to change the backend of matplotlib. To do so, go in your home directory and in this directory

```
/home/user/.config/matplotlib
```

add a file called `matplotlibrc` with the following content:

```
backend: Agg
```

Save, exit the shell, start a new shell.

pysam

```
from pysam.libchtslib import *
...ImportError: libhts.so.1: cannot open shared object file: No such file or directory
```

This may be solved by removing conda installation and using pip instead:

```
conda remove pysam
pip install pysam
```
qt

```python
from PyQt5.QtWebKitWidgets import QWebView
...
```

This may be solved by re-installation qt using the main anaconda channel (instead of bioconda):

```
conda install --override-channels -c anaconda qt
```

**Expected input format**

Most of the pipelines and standalone expect FastQ files with the extension `fastq.gz` meaning that files are gzipped. Besides, the filename convention is as follows:

```
PREFIX_R1_.fastq.gz
```

that is `_R1_` and `_R2_` indicates the paired or single-ended files and the PREFIX is used to create directories or reports; it must be present.

**Changelog**

**Contents**

- *Changelog*
  - 0.1.22 - March 2017
  - 0.1.21 - Feb 2017
  - 0.1.20 - Feb 2017
  - 0.1.19 - Feb 2017
  - 0.1.17/0.1.18 - Jan 2017
  - 0.1.16
  - 0.1.15
  - 0.1.14
  - 0.1.13
  - 0.1.12
  - 0.1.11
  - 0.1.10 - July 2016
  - 0.1.7 to 0.1.9 - July 2016
  - 0.1.6 - June 2016
  - 0.1.5 June 2016
  - 0.1.4
0.1.22 - March 2017

• CHANGES:
  – Sequanix/Sequana: - config file can have the yml extension (in addition to yaml)

0.1.21 - Feb 2017

• NEWS:
  – add sequana_debug_level function at top level to switch verbosity of informative messages (default is WARNING).
  – add pacbio module #351
  – quality control pipeline: atropos can be used in place of cutadapt #346

• CHANGES:
  – Running Median is 10 times faster #345
  – sequana_coverage: (1) –file1 alone was not working (2) automatically copy cluster-config in working directory and update runme.sh accordingly #342
  – sequana standalone:
    * handles cluster_config Snakemake option
    * add error message when adapter name is incorrect
  – sequanix: the help dialog is now created inside designer and has a proper scrollable browser dialog. cluster_config Snakemake option is also handle.
  – Remove galleria JS lib and related files (htmltools)
  – sequana_coverage: add –logging-level option

• BUG:
  – Fix #352 : allow gc window size to be even (warning is shown and +1 to window size)
  – Fix # 354: cutadapt report that was mixing up R1/R2 trimming in the images.
  – --output-directory in sequana_coverage was failing
  – in coverage, centralness was buggy (regression) and use number of ROIs instead of the total base length #347
  – Fix multi_report summary for single end case #349

0.1.20 - Feb 2017

• CHANGES:
  – remove pyquickhelper dependencies and add a simple rest2html function in misc module.
0.1.19 - Feb 2017

- **CHANGES:**
  - **misc module:** factorise `on_cluster()` function used in compressor scripts to be used in other tools such as sequanix
  - **compressor:** limits max number of jobs to 20 (can be bypass manually), prevent run on TARS if snakemake-cluster not provided.
  - **rules:**
    * dag: now the snakemake is called inside a temporary directory to avoid clash with the current snakemake process. This avoid error message. Fixes https://github.com/sequana/sequana/issues/331
  - **__init__** was optimized as well as many modules to make use of the lazy import mechanism. The reporting package is not part of the exposed module. So:
    ```python
    from sequana import BAMReport
    ```
    is now:
    ```python
    from sequana.reporting.report_bam import BAMReport
    ```

- **NEWS:**
  - Sequanix stable version
  - add TrueSeq adaptors
  - add lazy import mechanism to speed up the time to import sequana, which speeds up the --help in the standalone

0.1.17/0.1.18 - Jan 2017

Main **NEWS** The GUI was completed and the current pipelines stabilised (RNA-seq, quality control, variant calling). The test suite was switched from nosetests to pytest, in particular to perform tests more easily on the Qt GUI.

- **BUG Fixes:**
  - experimental design and adapters API simplified fixing a few bugs in the process. Doc and tested finalised.
  - Fix cutadapt rules, which was not filling the fwd and rev properly anymore when using the design file.
  - in sequana main script, --reference was used by quality_pipeline only. Now, available for all.
  - Fix the main script for the reference in variant calling pipeline.

- **CHANGES:**
  - **sequana_compressor:** for conversion from e.g gz to bz2, use a pipe instead of double IO. Updated docs and tests ready for production.
  - sequana standalone: - --pattern changed to --input-pattern - --output-directory changed to --working-directory
  - remove pipetools module (obsolet)
  - GUI revisited with qt designer + can now also read any snakefile/config file combo (not just sequana pipelines)
– RULES: adapters can now use adapter_type without a design (fwd and rev gets filled automatically)

• NEWS:
  – add rubicon adapters
  – add ability to read JSON in SequanaConfig

0.1.16

• BUG Fixes:
  – Fix sequana_taxonomy (https://github.com/sequana/sequana/issues/308)
  – Fix typo in sequana_coverage for multiple chromosome (https://github.com/sequana/sequana/issues/307)

• NEWS:
  – SequanaConfig can read back a SequanaConfig instance
  – Added a DummyManager for minimalist manager to create reports

0.1.15

• CHANGES:
  – coverage: https://github.com/sequana/sequana/issues/302 add histogram, better stats table. add –output-directory
  – Update docker (add bowtie, subread, firefox)
  – snaketools:
    * empty strings are kept as empty strings (not None)
    * remove check() method in SequanaConfig
    * cleanup (removing of templates) can be switch off

0.1.14

• CHANGES:
  – fastqc.histogram_sequence_lengths (log2 scale to log10)
  – multi_summary fixed and available for the quality_control pipeline
  – sequana_compressor: add –keep-going option by default so that if a file fails, other independent files are processed.
  – snaketools:
    * remove SnakeMakeProfile (not used)
    * remove sequana_check_config (not used)
    * remove deprecated __get_tagname
    * remove ExpandedSnakefile since not required anymore
    * Fix sample_file2 option that was not encoded properly
    * PipelineManager and SequanaConfig use new yaml parser
sequana_coverage:  – add back the sample name as prefix of the HTML report name – a BED with two coverage columns is now accepted – –download-genbank option added

– sequana_summary works for the quality_control pipeline
– Simplify combos of input_directory, input_pattern, input_samples, the new possible mutually exclusive
input parameters of sequana standalone and all pipelines.

• BUGS:
  – Kraken: if no reads classified at all, errors were raised and quality_control summary report would fail. This is fixed now with a “nodata” image being shown.

• NEWS
  – GUI (draft version)
  – fq.gz are now allowed in the pipelines and should be supported in the future
  – More tests in particular a ./test/pipelines/ new directory

0.1.13

• CHANGES:
  – revisited all pipelines so that they can work of multi samples.
  – quality_phix, quality_taxon pipelines merged in quality_control pipeline
  – running meadian won’t fail anymore with odd window size (we add +1)
  – rulegraph is used as well as dag to create figures of the pipelines

• NEWS:
  – compressor: includes dsarc format in addition to bz2 and gz
  – snakemake rule extension for sphinx
  – add a pipeline manager in snaketools to handle all pipelines
  – a designexp module to handle adapter design files

0.1.12

• BUGS:
  – Fix bug in cutadapt pipeline when there is no adapters. Force a dummy adapters (XXX) otherwise trimming is performed on read1 only

• NEWS:
  – compressor rule and script available.
  – coverage annotation
  – multiple_summary draft
0.1.11

• NEWS:
  – add a docker
  – sequana_summary standalone
  – sequana_mapping standalone
  – Module has an overview field

• BUG FIXES:
  – cutadapt report handles single-end tables. Fix the reverse complement adapter files for the paired-end case

• CHANGES:
  – sequana_standalone: final version with stats

0.1.10 - July 2016

• NEWS:
  – sequana_coverage standalone
  – de-novo pipeline

• CHANGES:
  – Remove AdapterDB, a draft version that uses Kraken to detect adapters. Not relevant anymore
  – config.yaml is now in each pipeline to have a simplified version
  – sequana can known use single_indexed or multiple_indexed adapters, which are also provided within sequana (Nextera and PCR free cases)
  – Release for production (quality_taxon pipeline)

0.1.7 to 0.1.9 - July 2016

• NEWS:
  – rule data added and used in phix_removal (fastq_sampling + raw data switch)
  – kmer module
  – sequana_taxonomy standalone

• CHANGES:
  – reports are now in ./sequana/reporting
  – MAJOR refactoring of report/ directories in all pipelines to make them independent from the temporary analysis, which can then be removed.

• BUGS:
  – Fix running median issue in bedtools (window size larger than contig size)
0.1.6 - June 2016

• NEWS:
  – KrakenDownload class: download kraken_toydv from sequana/data repository or minikraken into a local directory
  – New method in FastQC to show ACGT content
  – Genomecov renamed into GenomeCov
  – Update main script significantly to create multiruns and handle adapters
  – GC content and plot GC vs coverage added in GenomeCov

• CHANGES:
  – sequana_data by default looks into resources/testing directory
  – in fastq module: FastQC a bit faster and FastQRandom class removed
  – add a moving_average function in misc module

• BUGS:
  – sequana_data was showing __init__ and __pycache__ as possible data sets
  – databases: filelist as a list was not implemented
  – in fastq.FastQ extra_head in gzip mode was missing the last row

0.1.5 June 2016

• NEWS
  – sequana_taxonomy standalone available (kraken + krona)
  – sequana standalone available
  – quality_taxon pipeline available
  – module coverage for theoretical computations

• CHANGES:
  – module vcf_to_snpeff renamed as snpeff
  – lots of doc added
  – clean adapters module

• BUG:
  – Fix bug in running median (shift)

0.1.4

• add gallery in the documentation
• remove old pipelines/ directory
• sequana standalone refactored (–init option added)
• Pipeline quality_taxon added
• Taxonomy rules included
• Fix the stats image creation

0.1.3

• NEWS
  – Update the pipeline phix_removal

0.1.1 - 0.1.2

• NEWS
  – Phix pipeline added

0.1.0 April 2006

• NEWS
  – rules in sequana/rules and pipelines in sequana/pipelines
  – standalone tool called sequana to download a Snakefile and config file
  – modules for post-analysis: bamtools, vcf_filter, fastq, ....
  – Fully tested (90% coverage) and documented
  – Set of reports

Glossary

BAI The index file for a file generated in the BAM format. (This is a non-standard file type.)
BAM Binary version of the Sequence Alignment Map (SAM) format.
BED Format that defines the data lines displayed in an annotation track.
DSRC A compression tool dedicated to FastQ files
FASTA FASTA-formatted sequence files contain either nucleic acid sequence (such as DNA) or protein sequence information. FASTA files store multiple sequences in a single file.
GFF General Feature Format, used for describing genes and other features associated with DNA, RNA and Protein sequences.
Module A directory that contains a snakemake rule and an associated README file. This is especially relevant for the Sequana pipelines. See Developer guide.
SAM Sequence Alignment Map is a generic nucleotide alignment format that describes the alignment of query sequences or sequencing reads to a reference sequence or assembly
Snakefile A file that contains one or several Snakemake rules
VCF Variant Call Format, for use with the variant calling pipeline
Indices and tables

- genindex
- modindex
- search
Python Module Index

a
sequana.adapters, 45

b
sequana.bamtools, 50
sequana.bedtools, 57

c
sequana.coverage, 63

d
sequana.databases, 66
sequana.datatools, 107

e
sequana.expdesign, 68

f
sequana.fastq, 70
sequana.freebayes_vcf_filter, 105

i
sequana.iotools, 76

k
sequana.kmer, 76
sequana.kraken, 77
sequana.kraken_builder, 82
sequana.krona, 81

m
sequana.misc, 103

p
sequana.pachio, 84
sequana.phred, 89

r
sequana.reporting.report_bam, 107
sequana.reporting.report_main, 107
sequana.reporting.report_summary, 107
sequana.resources.data, 108
sequana.running_median, 91

s
sequana.snakertools, 94
sequana.snpeff, 102
sequana.stats, 104

t
sequana.tools, 104

119
Index

A
a (Coverage attribute), 64
accession_to_info() (EUtilsTools method), 67
Adapter (class in sequana.adapters), 45
adapter_removal_parser() (in module sequana.adapters), 49
AdapterReader (class in sequana.adapters), 46
adapters_to_clean_ngs() (in module sequana.adapters), 49
add_locus_in_fasta() (Snpeff method), 102
add_urls() (DOTParser method), 95
Alignment (class in sequana.bamtools), 55
all_extensions (FileFactory attribute), 97
as_dict() (Alignment method), 56

B
BAI, 117
BAM, 117
BAM (class in sequana.bamtools), 50
bam_analysis_to_json() (BAM method), 50
bam_to_mapped_unmapped_fastq() (in module sequana.tools), 104
BAMPacbio (class in sequana.pacbio), 84
basenames (FileFactory attribute), 97
BED, 117
bed (ChromosomeCov attribute), 60
boxplot_quality() (FastQC method), 73
build_kmer() (in module sequana.kmer), 76

C
check() (ExpDesignBase method), 69
check() (FindAdaptersFromDesign method), 49
check() (Module method), 98
check_sequana_fields() (SequanaConfig method), 102
ChromosomeCov (class in sequana.bedtools), 59
circular (GenomeCov attribute), 58
clean_db() (KrakenBuilder method), 83
cleanup() (SequanaConfig method), 102
cleanup_config() (SequanaConfig method), 102
cluster_config (Module attribute), 98
columns() (ChromosomeCov method), 60
comment (Adapter attribute), 46
comments (AdapterReader attribute), 47
compute_frequency() (in module sequana.freebayes_vcf_filter), 107
compute_ge_content() (GenomeCov method), 58
compute_strand_balance() (in module sequana.freebayes_vcf_filter), 107
compute_zscore() (ChromosomeCov method), 60
copy() (Module attribute), 98
copy_cluster (Module attribute), 98
copy_requirements() (SequanaConfig method), 102
count_lines() (FastQ method), 71
count_reads() (FastQ method), 71
count_reads_gz() (FastQ method), 71
Coverage (class in sequana.coverage), 63

description (Module attribute), 98
df (BAMPacbio attribute), 84
df (Filtered_freebayes attribute), 105
df (KrakenResults attribute), 78
DOTParser (class in sequana.snaketools), 94
DoubleThresholds (class in sequana.bedtools), 63
download() (KrakenDownload method), 81
download_accession() (ENADownload method), 67
download_accession() (KrakenBuilder method), 83
download_archaea() (ENADownload method), 67
download_archaealvirus() (ENADownload method), 67
download_bacteria() (ENADownload method), 67
download_fasta() (ENADownload method), 67
download_fasta_and_genbank() (in module sequana.snpeff), 103
download_list() (ENADownload method), 67
download_macaca() (ENADownload method), 67
download_organelle() (ENADownload method), 67
download_phage() (ENADownload method), 67
download_plasmids() (ENADownload method), 67
download_taxonomy() (KrakenBuilder method), 83
download_viriod() (ENADownload method), 67
download_viruses() (ENADownload method), 67
download_viruses() (KrakenBuilder method), 83
DSRC, 117
dv (KrakenDownload attribute), 81

e
ena_id_to_gi_number() (ENADownload method), 67
ENADownload (class in sequana.databases), 66
error() (PipelineManager method), 100
EUtilsTools (class in sequana.databases), 67
evenness() (in module sequana.stats), 104
ExpDesignAdapter (class in sequana.expdesign), 68
ExpDesignBase (class in sequana.expdesign), 69
ExpDesignHiSeq (class in sequana.expdesign), 69
ExpDesignMiSeq (class in sequana.expdesign), 68
extension (SnpEff attribute), 102
extensions (FileFactory attribute), 97
extract_head() (FastQ method), 71

F
FASTA, 117
fasta_fwd_rev_to_columns() (in module sequana.adapters), 49
FastQ (class in sequana.fastq), 70
FastQC (class in sequana.fastqc), 72
FastQFactory (class in sequana.snaketools), 95
feature_dict (GenomeCov attribute), 59
FileFactory (class in sequana.snaketools), 96
filenames (FileFactory attribute), 97
filter() (FastQ method), 71
filter_length() (BAMPacbio method), 84
filter_vcf() (VCF_freebayes method), 106
Filtered_freebayes (class in sequana.freebayes.vcf_filter), 105
filters_params (VCF_freebayes attribute), 106
FindAdaptersFromDesign (class in sequana.adapters), 48
findpos() (in module sequana.misc), 103

G
G (Coverage attribute), 64
gc_window_size (GenomeCov attribute), 59
genbank_filename (GenomeCov attribute), 59
GenomeCov (class in sequana.bedtools), 57
get_actg_content() (FastQC method), 71
get_adapter_by_identifier() (AdapterReader method), 47
get_adapter_by_index_name() (AdapterReader method), 47
get_adapter_by_index_seq() (AdapterReader method), 47
gene_adapter_by_sequence() (AdapterReader method), 47
get_adapters_from_sample() (FindAdaptersFromDesign method), 49
get_args() (DoubleThresholds method), 63
get_centralness() (ChromosomeCov method), 61
get_cv() (ChromosomeCov method), 61
get_df() (ChromosomeCov method), 61
get_evenness() (ChromosomeCov method), 61
g_file1() (FastQFactory method), 96
g_file2() (FastQFactory method), 96
g_flags() (BAM method), 50
g_flags() (SAMFlags method), 57
g_flags_as_df() (BAM method), 50
g_full_stats_as_df() (BAM method), 51
g_gaussians() (ChromosomeCov method), 61
g_get_gi() (BAM method), 51
g_get_gi() (ChromosomeCov method), 61
g_get_gi() (KrakenBuilder method), 83
gkm() (in module sequana.kmer), 76
g_length() (BAM method), 51
g_mapped_read_length() (BAM method), 51
g_mapq() (BAM method), 51
g_max_gi() (ChromosomeCov method), 61
g_mean_contig_length() (Coverage method), 65
g_mean_cover() (ChromosomeCov method), 61
g_mean_number_contigs() (Coverage method), 65
g_mean_reads_per_contig() (Coverage method), 65
g_meaning() (SAMFlags method), 57
g_metrics_count() (BAM method), 51
g_percent_genome_sequence() (Coverage method), 65
g_read_names() (BAM method), 51
g_required_coverage() (Coverage method), 65
g_roi() (ChromosomeCov method), 61
g_samflags_count() (BAM method), 51
g_sample() (FindAdaptersFromDesign method), 49
g_sequana_adapters() (in module sequana.adapters), 49
g_size() (ChromosomeCov method), 61
g_stats() (BAM method), 52
g_stats() (ChromosomeCov method), 61
g_stats() (FastQC method), 74
g_stats() (GenomeCov method), 59
g_summary() (Coverage method), 66
g_table() (Coverage method), 66
g taxpayer() (KrakenResults method), 78
g_taxons_from_gi() (KrakenBuilder method), 83
gget() (ChromosomeCov method), 62
ggetlogdir() (PipelineManager method), 100
ggetname() (PipelineManager method), 100
ggetrawdata() (PipelineManager method), 100
ggetreportdir() (PipelineManager method), 100
gwkdir() (PipelineManager method), 100
GFF, 117

H
hdtr (DoubleThresholds attribute), 63
high (DoubleThresholds attribute), 63
high2 (DoubleThresholds attribute), 63
hist() (GenomeCov method), 59

Index
hist_GC() (BAMPacbio method), 84
hist_len() (BAMPacbio method), 86
hist_snv() (BAMPacbio method), 87
hist_ZMW_subreads() (BAMPacbio method), 85
histogram_gc_content() (FastQC method), 74
histogram_sequence_coordinates() (FastQC method), 74
histogram_sequence_lengths() (FastQC method), 74

I
identifier (Adapter attribute), 46
Identifier (class in sequana.fastq), 70
identifiers (AdapterReader attribute), 48
imshow_qualities() (FastQC method), 75
index_names (AdapterReader attribute), 48
index_sequence (Adapter attribute), 46
index_sequences (AdapterReader attribute), 48
init() (KrakenBuilder method), 83
is_executable() (Module method), 99
is_pl pipeline() (Module method), 99
iter_mapped_reads() (BAM method), 52
iter_unmapped_reads() (BAM method), 52

J
joining() (FastQ method), 71

K
kraken_to_csv() (KrakenResults method), 78
kraken_to_json() (KrakenResults method), 78
kraken_to_krona() (KrakenResults method), 78
KrakenAnalysis (class in sequana.kraken), 80
KrakenBuilder (class in sequana.kraken_builder), 82
KrakenDownload (class in sequana.kraken), 81
KrakenPipeline (class in sequana.kraken), 79
KrakenResults (class in sequana.kraken), 77
KronaMerger (class in sequana.krona), 81

L
L (Coverage attribute), 64
launch_snpeff() (SnpEff method), 102
ldtr (DoubleThresholds attribute), 63
low (DoubleThresholds attribute), 63
low2 (DoubleThresholds attribute), 63

M
mean_quality (Quality attribute), 91
Module, 117
Module (class in sequana.snaketools), 97
modules (in module sequana.snaketools), 102
moving_average() (ChromosomeCov method), 62
moving_average() (in module sequana.stats), 104

N
N (Coverage attribute), 64
n_lines (FastQ attribute), 71
n_reads (FastQ attribute), 71
name (Adapter attribute), 46
name (Module attribute), 99
nb_pass (BAMPacbio attribute), 88
next() (FastQ method), 71

O
on_cluster() (in module sequana.misc), 104
onweb() (Module method), 99
overview (Module attribute), 99

P
path (Module attribute), 99
pathname (FileFactory attribute), 97
pathnames (FileFactory attribute), 97
PipelineManager (class in sequana.snaketools), 99
plot() (KrakenResults method), 78
plot() (Quality method), 91
plot() (SnakeMakeStats method), 101
plot_acgt_content() (FastQC method), 75
plot_and_save() (SnakeMakeStats method), 101
plot_bar_flags() (BAM method), 52
plot_bar_mapq() (BAM method), 53
plot_coverage() (ChromosomeCov method), 62
plot_gc_content() (BAM method), 54
plot_GC_read_len() (BAMPacbio method), 88
plot_gvcov() (ChromosomeCov method), 62
plot_hist_coverage() (ChromosomeCov method), 62
plot_hist_normalized_coverage() (ChromosomeCov method), 62
plot_hist_zscore() (ChromosomeCov method), 62

Q
Quality (class in sequana.phred), 90
quality (Quality attribute), 91

R
random() (FastQ method), 71
read() (ExpDesignBase method), 69
readme (Module attribute), 99
realpaths (FileFactory attribute), 97
record (Variant attribute), 107
requirements (Module attribute), 99
reset() (BAMPacbio method), 89
rest2html() (in module sequana.misc), 103
resume (Variant attribute), 107
reverse() (AdapterReader method), 48
reverse_complement() (AdapterReader method), 48
rewind() (FastQ method), 71
rewind() (VCF_freebayes method), 106
run() (KrakenAnalysis method), 81
run() (KrakenBuilder method), 84
run() (KrakenPipeline method), 80
run() (RunningMedian method), 94
running_median() (ChromosomeCov method), 62
running_median() (in module sequana.running_median), 94
RunningMedian (class in sequana.running_median), 91

S
SAM, 117
SAMFlags (class in sequana.bamtools), 56
sample_names (FindAdaptersFromDesign attribute), 49
save() (SequanaConfig method), 102
save_adapters_to_fasta() (FindAdaptersFromDesign method), 49
select_random_reads() (FastQ method), 71
sequana.adapters (module), 45
sequana.bamtools (module), 50
sequana.bedtools (module), 57
sequana.coverage (module), 63
sequana.databases (module), 66
sequana.datatools (module), 107
sequana.expdesign (module), 68
sequana.fastq (module), 70
sequana.freebayes_vcf_filter (module), 105
sequana.iotools (module), 76
sequana.kmer (module), 76
sequana.kraken (module), 77
sequana.kraken_builder (module), 82
sequana.krona (module), 81
sequana.misc (module), 103
sequana.pacbio (module), 84
sequana.phred (module), 89
sequana.reporting.report_bam (module), 107
sequana.reporting.report_main (module), 107
sequana.reporting.report_summary (module), 107
sequana.resources.data (module), 108
sequana.running_median (module), 91
sequana.snaketools (module), 94
sequana.snpeff (module), 102
sequana.stats (module), 104
sequana.tools (module), 104
sequana.data() (in module sequana.datatools), 107
SequanaConfig (class in sequana.snaketools), 101
sequence (Adapter attribute), 46
sequences (AdapterReader attribute), 48
set_gaussians() (ChromosomeCov method), 62
show() (KrakenPipeline method), 80
Snakefile, 117
snakefile (Module attribute), 99
SnakeMakeStats (class in sequana.snaketools), 100
Snpeff (class in sequana.snpeff), 102
split_chunks() (FastQ method), 72
split_lines() (FastQ method), 72
StatsBAM2Mapped (class in sequana.tools), 104
strand_ratio() (in module sequana.freebayes_vcf_filter), 107
stride() (BAMPacbio method), 89
switch_header_to_gi() (ENADownload method), 67

t

taxons (KrakenResults attribute), 79
textwrap() (in module sequana.misc), 103
to_csv() (ChromosomeCov method), 62
to_csv() (Filtered_freebayes method), 105
to_csv() (GenomeCov method), 59
to_dict() (AdapterReader method), 48
to_fasta() (AdapterReader method), 48
to FASTQ (BAM method), 55
to_html() (StatsBAM2Mapped method), 104
to_json() (KrakenResults method), 79
to_kmer_content() (FastQ method), 72
to_krona() (FastQ method), 72
to_tsv() (KronaMerger method), 82
to VCF (Filtered_freebayes method), 105

V

Variant (class in sequana.freebayes_vcf_filter), 106
variants (Filtered_freebayes attribute), 105
VCF, 117
VCF_freebayes (class in sequana.freebayes_vcf_filter), 105

W

wget() (in module sequana.misc), 103
window_size (GenomeCov attribute), 59

Y

YamlDocParser (class in sequana.iotools), 76