KvarQ Documentation

Release 0.12.2

Andreas Steiner

October 17, 2014

CHAPTER 1

Contents

1.1 Installing KvarQ

1.1.1 Precompiled packages

The different releases are available als compiled packages for Microsoft Windows and OSX (10.6 and later): http://github.com/kvarq/kvarq/releases

Note that the packages are currently **not signed** and you therefore have to enable OS X to run programs from unidentified developers if you run OS X 10.8 or newer.

1.1.2 Installing KvarQ From Source

The source code is hosted in a git repository at http://github.com/kvarq/kvarq

Dependencies

KvarQ does not have any external dependencies, apart from **Sphinx_** for building the html documentation from the `docs/*.rst` sources files.

Linux

in case your system runs a python older than version **2.7**, you have to install a newer version of python first; this is easiest done locally

```
wget http://www.python.org/ftp/python/2.7.4/Python-2.7.4.tgz
tar xzf Python-2.7.4.tgz
cd Python-2.7.4
mkdir $HOME/py
./configure --prefix=$HOME/py
make && make install
PATH=$HOME/py/bin:$PATH
export $PATH
python -V
```

then download and install setuptools:

```
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py -O - | python
```

download the latest source distribution and build (calling `setup.py` with `test` will also copy the compiled library into the source directory)

```
wget https://github.com/kvarq/kvarq/archive/master.zip
unzip master.zip
rm master.zip
cd kvarq-master
python setup.py test
```

now you can **either install** KvarQ (optionally into a virtual environment):

```
python setup.py install
```

or setup an alias after including KvarQ into your `PYTHONPATH`. This is the method of choice if you intend to plan *to modify the KvarQ source* because you don't need to make a fresh installation after every change – but don't forget to re-run `python setup.py test` in case you changed the C source code to make sure the compiled extension is copied into the correct directory (or by sourcing the script `. ./activate`)

```
PYTHONPATH=`pwd`; export PYTHONPATH
alias kvarq='python -m kvarq.cli'
kvarq -h
```

In either way, you now have the KvarQ command at your disposal and can continue *using the commandline* or start the *graphical user interface*.

OS X

Prerequisites:

- If you use OS X Snow Leopard (10.6) or below, you first have to install Python 2.7 (this version of python is included in OS X Lion 10.7 and newer; but you might nevertheless want to install a vanilla copy of Python)
- On the other hand, OS X Snow Leopard and older include a C compiler that is needed to build the program. If you have no C compiler installed (you get a `command not found` error when you type `gcc` or `clang` at the Terminal), you need to download Xcode from Apple's developer page (registering an account only takes some minutes). Choose **Command Line Tools for Xcode** from the "Developer Tools" category.

From this point on, follow the steps outlined in the *Linux section*. If you want to create an OS X application, you will also need to download and install py2app.

Windows

Prerequisites:

- First download and install Python (at least version 2.7). You should download the **32bit** version regardless of your machine architecture (or you will run into problems with the steps outlined below). If you plan to use python for scientific ends, you might want to install the Enthought Canopy Distribution that bundles many interesting packages.
- Because KvarQ uses a compiled module to scan through the files you will have to install a C compiler. The simplest choice is to download and install Microsoft Visual Studio Express (e.g. VS Express 2012). This will automatically set the environment variable `VSxx0COMNTOOLS` (with `xx` being the version of visual studio).
- KvarQ includes a pthreads in `win32/pthreads` for compiling the C extension. You have to **copy** this file into your windows folder or make sure that `win32/pthreads` is in your DLL search path.

You should now be able to download, build and test the program pretty much the same way as *described above*. To create a stand-alone executable package (via `python setup.py py2exe`) you will also need to download

py2exe. Finally, you will probably want to install some packaging system (not installed by default) to get more python packages.

1.2 Using KvarQ Command Line Interface (CLI)

1.2.1 Using The Command Line

Depending on which *installation instructions* you followed, the KvarQ command line utility will be accessible in a different way

- installation from source: simply enter KvarQ on the command line or alternatively call `python -m kvarq.cli` (make sure the directory containing `kvarq/` is in the `PYTHONPATH` if you just downloaded & compiled KvarQ without installing it)
- binary installation windows: go to the directory with the kvarq files and start `kvarq.exe`
- binary installation OS X: enter the following command in a shell: `/path/to/kvarq.app/Contents/MacOS/python -m kvarq.cli`

In either case you can use all the functionality described below by using this one comand with appopriate flags. The different flags are all described briefly if KvarQ is run with the `--help` command line switch.

Note that most command line arguments apply to a specific **subcommand**, but some arguments (such as declaration of additional *testsuites* directories using the `-t` option or logging using the `-d` and `-l` options) apply to all commands and are therefore specified *before* the subcommand.

1.2.2 Loading of Testsuites

Loading of *Testsuites* is a two step process:

1. First all available testsuites are **discovered**. KvarQ looks for testsuites in the following directories
   (a) The directory `testsuites/` in the directorythat contains the executable. That is where the testsuites are originally located after download.
   (b) In the directory `kvarq_testsuites/` in the user's home directory.
   (c) In the directory `testsuites/` in the current working directory
   (d) From directories specified with the environment variable `KVARQ_TESTSUITES` – use your system's path separator (`;` on windows, `:` on most other systems).
   (e) From any directories specified with the general `--testsuite-directory` (shorthand `-t`) command line switch.

   If a testsuite is found several times, the last occurrence is used. This allows for easy modification of existing testsuites: simply copy the files into the directory `./testsuites/` and modify them. Because discovery takes place later on the current working directory, these modifications will override the original testsuites. This mechanism also easily allows to have different versions of the same testsuite – simply rename the directory (e.g. copying `MTBC/` to `MTBC-legacy/` before applying any incompatible changes to the testsuites).
2. Testsuites are later on loaded from this pool of discovered testsuites when necessary. When *scanning a .fastq file*, the testsuites have to be specified explicitly, but for most other actions (such as *showing results*), testsuites are loaded automatically.

1.2.3 Scanning a File

The `scan` subcommand scans a `.fastq` file and saves the results in a `.json` file. There are many additional parameters to this command. See the following examples to illustrate some scenarios.

Simplest scenario: Scan a file, showing a progress bar during the scanning process and save the results (using the whole *MTBC testsuites*):

```
kvarq scan -l MTBC -p H37v_strain.fastq H37v_strain.json
```

Being more verbose and copying the log output into a separate file:

```
kvarq -d -l kvarq.log scan -l MTBC -p H37v_strain.fastq H37v_strain.json
```

In the following example, only the phylogenetic testsuite is loaded:

```
kvarq scan -l MTBC/phylo H37v_strain.fastq H37v_strain.json
```

There are many more command line options; in the following example, KvarQ uses only one thread (this results in a much slower scanning, but would be advisable if many scans are executed in parallel as scheduled jobs), specifies explicitly the `.fastq` variant (normally this variant is guessed by peeking into the quality scores), and ignores all reads that are shorter than 30 base pairs (after quality trimming):

```
kvarq scan -l MTBC -t 1 --variant Sanger -r 30 -p H37v_strain.fastq H37v_strain.json
```

During the scanning, it is possible to obtain some additional statistics by pressing `<CTRL-C>` on the terminal (this does not work on Windows when you use a MinGW bash prompt). Pressing `<CTRL-C>` twice within two seconds will interrupt the scanning process and proceed to calculate the results with the data gathered so far.

Usually, the default parameters for quality cut-off and minimum overlap (see *Configuration Parameters*) work pretty well. If you encounter problems with a particular `.fast` file, refer to the example in *Determine Scanning Parameters*.

1.2.4 Extracting results from a batch of scans

Normally, you would run KvarQ over a whole series of `.fastq` files and then in the end extract the relevant information from the resulting `.json` files. The `summarize` command allows such an extraction of summary information from multiple `.json` files. The following command extracts the results, as reported by the different testsuites, and saves it to a `.csv` file:

```
kvarq summarize results/*.json > results.csv
```

1.2.5 Showing information about testsuites

The `info` commands displays version information and some summary statistics about testsuites. Testsuites can be specified the same way as when *scanning a file*, so this command is handy to estimate how many templates would be loaded with a given testsuite selection. Using the `-L` command line switch loads all discovered testsuites:

```
kvarq info -l MTBC
kvarq info -L
```

1.2.6 Directly Analysing a .fastq

Use to `show` subcommand to analyze `.fastq` files directly without performing a scan of the file. For example, the readlengths that would result from a specified quality cutoff can be displayed using:

```
kvarq show -Q 13 H37v_scan.fastq
```

1.2.7 Showing Results

Some simple analysis of `.json` files are possible using the command line, but the *GUI explorer* is much more powerful.

The subcommand `illustrate` can be used to show the final results of the scanning, as well as detailed information about the coverages or a histogram of the (quality-cut) readlengths encountered:

```
kvarq illustrate -r H37v_strain.json
kvarq illustrate -c H37v_strain.json
kvarq illustrate -l H37v_strain.json
```

1.2.8 Updating Results

Since a `.json` file contains not only the final results but also the intermediare results (encoded in `kvarq.analyse.Coverage`), it is possible to update the results sections after modifying the code without having to re-scan the `.fastq` file. The `.json` file is updated in-place:

```
kvarq -d update H37v_scan.json
```

1.2.9 More Usage Examples

Verify File Format Integrity

Check all `.fastq` files in a directory structure for file format integrity

```
#!/bin/bash
for fastq in `find /tbresearch -name \*.fastq`; do
  python -m kvarq.cli -d show "$fastq" 2>"$0_error.log"
  err="$?"
  echo $err $fastq
  if [ $err -ne 0 ]; then
    # file format error
    base=`basename "$fastq"`
    mv "$0_error.log" "${base%.fastq}.log"
  fi
done
rm "$0_error.log"
```

Determine Scanning Parameters

To find ideal values for the *Configuration Parameters* it's a good idea to first have a look at the output of `python kvarq.cli show -Q 13` (minimum PHRED score of 13 corresponds to p<0.05).

In the following example, the quality score needs to be lowered to yield anything useable from the `.fastq`:

```
[   0-   3] 4440 (44%)**************************************************************
[   3-   6] 2995 (29%)*****************************************
[   6-   9] 1221 (12%)*****************
[   9-  12]  618 ( 6%)*********
[  12-  15]  364 ( 3%)*****
```

```
[  15-  18]  206 ( 2%)***
[  18-  21]   93 ( 0%)*
[  21-  24]   44 ( 0%)
[  24-  27]   11 ( 0%)
[  27-  30]    1 ( 0%)
[  30-  33]    0 ( 0%)
[  33-  36]    0 ( 0%)
[  36-  39]    2 ( 0%)
[  39-  42]    1 ( 0%)
[  42-  45]    2 ( 0%)
[  45-  48]    2 ( 0%)
```

In the next example, the minimum overlap and minimum readlength should be adapted to something below 25:

```
[   0-   2]  183 ( 1%)******
[   2-   4]  209 ( 2%)*******
[   4-   6]  611 ( 6%)*********************
[   6-   8]  839 ( 8%)*******************************
[   8-  10]  896 ( 9%)*********************************
[  10-  12]  822 ( 8%)*****************************
[  12-  14]  867 ( 8%)*******************************
[  14-  16]  633 ( 6%)**********************
[  16-  18]  692 ( 6%)************************
[  18-  20]  628 ( 6%)**********************
[  20-  22]  499 ( 5%)*****************
[  22-  24]  520 ( 5%)******************
[  24-  26] 1810 (18%)*****************************************************************
[  26-  28]  706 ( 7%)************************
[  28-  30]   82 ( 0%)**
```

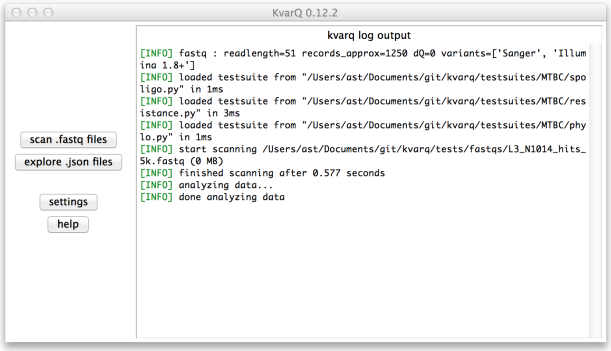
1.3 Using KvarQ Graphical User Interface (GUI)

1.3.1 Launching the GUI

Depending on *how KvarQ was installed*, there are different ways of launching the GUI

- Installation from source: simply enter `kvarq gui` on the command line or alternatively call `python -m kvarq.cli gui`. When KvarQ is launched this way, you can use some *command line switches* (for example specify the directory containing the *testsuites*).
- Binary installation windows: go to the directory with the KvarQ files and start `kvarq-gui.exe`
- Binary installation OS X: open the KvarQ application in the Finder

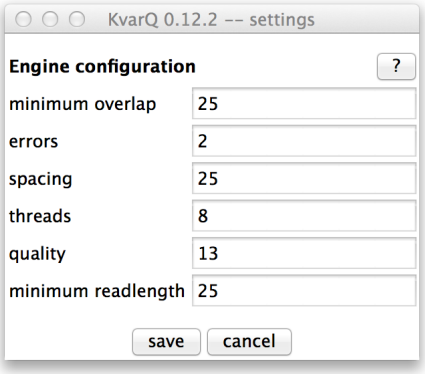
Below is what you should see after launching the GUI



The right pane in the main window shows the log output that describes the general activity as well as useful additional information during the scanning process. Important messages (warnings, errors) are highlighted in red.

The two buttons on the left open the *scanner* to scan `.fastq` files and the *explorer* to view results saved as `.json` files from previous scans.

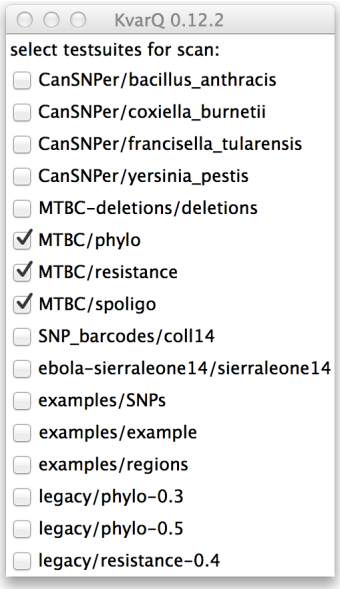
1.3.2 Configuring KvarQ



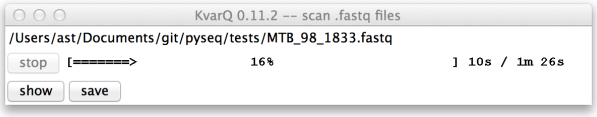
The *engine configuration parameters* can be modified in the settings window. Usually, the default values work well, but in some cases (such as old low-quality files) it can be advantageous to change some of these values.

1.3.3 The Scanner

This simple window allows to scan a single or multiple `.fastq` files to generate `.json` files (depending on whether a single or multiple files are selected in the file selection dialog).



When the selection of `.json` is done, the scanner shows a window with a list of *discovered testsuites* that can be checked individually to be included during the scan of the `.fastq` files.



The progress bar (yes, the design is on purpose to remind users to *use the command line*) shows the progress as well as estimated time left for the **current** file. The scanning process can be interrupted by clicking on the `stop` button.

Once the scanning process is done (for all files), the results can be saved to `.json` files (one per `.fastq` file). The result of the last scan can also be viewed directly, without saving it to file.

1.3.4 The Explorer

The explorer is a simple Tk program consisting of different windows:

Interpreting Coverages

KvarQ displays the results of the scanning process in the form of **coverages**. This display shows information about how many reads were mapped against the sequence of interest and whether there were any mutations detected. The same display is used for SNPs as well as for longer regions, althoug the signification of the displayed elements is somewhat different.

Keyboard Navigation

- switch between panes using `Tab`
- select item using `up`, `down`
- open window by pressing `enter`
- close window via `escape`

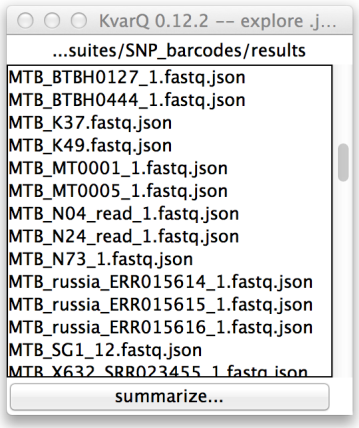


Figure 1.1: Directory explorer viewing `.json` files in a directory

Double-clicking any of the list items will open then `.json` explorer showing details on the selected file.
It is also possible to **export the analysis summary** of all displayed `.json` files to a excel sheet by using the button at the bottom of the list.

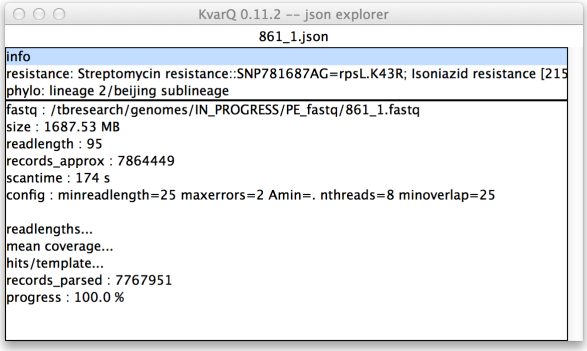


Figure 1.2: `.json` explorer showing general information about file

In the **upper pane** of the `.json` explorer shows an overview over the file. The contents of the **lower pane** depend on the selection in the upper pane. Because the **info** section is selected in the upper pane in this example, the lower pane shows general information about the scanning process, such as the scantime or the `kvarq.engine` configuration.
The items ending with `...` open another window when double-clicked (similar to the coverages described below).

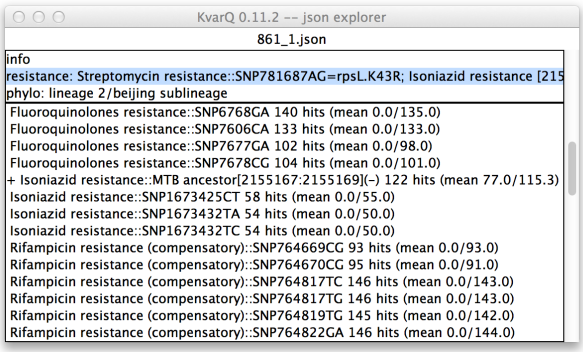


Figure 1.3: `.json` explorer showing analysis test details

In this case, the phylogenetic suite was selected in the **upper pane**. Therefore, all tests belonging to this testsuite are displayed in the lower pane. Every item in the upper pane (apart from the "info" item) consists of the testsuite name (in this case "phylo") and the summarized result (in this case lineage 2, sublineage bejing).

Every item in the **lower pane** informs about the following test details:

- Whether the test was "found positive" : a + sign in front of the test name signifies that this test was positive. For a SNP this means that the specified mutant allele was found and for a test covering a larger region of the genome this signifies that there was at least one mutation detected in the region of interest. A ~ sign (not shown) in front of the test name would mean that there were base calls with the most dominant base below 90%, suggesting a *mixed colony*.
- Test name that describes the genotype.
- Double semicolon `::` followed by description of what was tested (this can be a SNP or a region; regions are specified by their start/stop base position and a + or − specifying which strand is coding at this position).
- Double-clicking on an item in the lower pane opens a *coverage* window.

1.4 KvarQ testsuites

Testsuites define positions to scan for as well as how to interpret mutations. They have to be loaded (see *Loading Testsuites*) *selected prior to scanning* but also to analyze `.json` data using the *explorer*. A `.json` file generated with a certain combination of testsuites can only be analyzed using the explorer if testsuites with the same version are used (this is because `.json` files only contain the names of the SNPs but the location within genes is saved in the testsuites).

A KvarQ testsuite is a python source file that defines a `kvarq.genes.Testsuite` with the same name as the python file. Several of these testsuites can be grouped together within a single directory. Any number of such directories containing testsuite python files can be stored in a well defined location from which it is then *discovered in particular order*.

For example, the testsuites `spoligo`, `resistance`, and `phylo` are grouped together in the directory `MTBC/` and can be found in the directory `testsuites/` of KvarQ:

- `testsuites/MTBC/_util.py` : every file that starts with an underscore will **not** be loaded from KvarQ when loading testsuites from a directory, but can still be used from other python modules in the same directory via `from _util import ancestor` (which loads the hypothetical ancestor genome from a data file in the same directory
- `testsuites/MTBC/phylo.py` : scans for phylogenetic markers in MTBC
- `testsuites/MTBC/resistance.py` : tests for some common resistance mutations in MTBC
- `testsuites/MTBC/spoligo.py` : *in silico* spoligo typing of MTBC

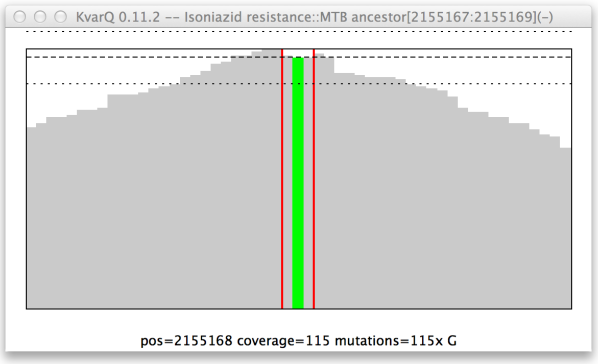


Figure 1.4: Mutation in the `katG` resistance confering codon.
General structure of a coverage window:

- The x axis is the genome **position**. Add the number showed on the x axis to the base position in parantheses in the figure title.
- The y axis is depth of **coverage**, piling all reads up that mapped to the given positions.
- The red vertical lines show start and stop of the **region of interest**. In this case, the region of interest is only three bases long, but 25 bases of spacers are added on either side when scanning for the region (see *Configuration Parameters*).
- The horizontal lines are mean and pseudo-variations of coverage over the region of interest.
- The colored graphs show **mutations**. In this example there is clearly a mutation that replaced the second base with a guanosine nucleotide. Note that not every read showed this mutation, but a handful had the original base (if every single read showed this mutation, the colored line would go all the way up to the thick black line).
- Moving the **mouse** over the graph shows quantitative information about the hovered genome position at the bottom of the graph.

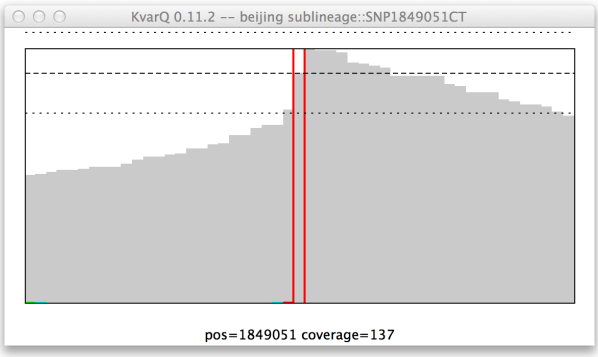


Figure 1.5: Coverage of a single nucleotide polyphormism (SNP), **mutant genotype**.
Because KvarQ is looking for a specific mutant sequence, the SNP is "found" if there is no mutation at its position, as is the case in this example (i.e. at position 157129 there is really a `T` and not a `C`).
Note: "No color" means mutant for SNP, while it means wild type for regions...

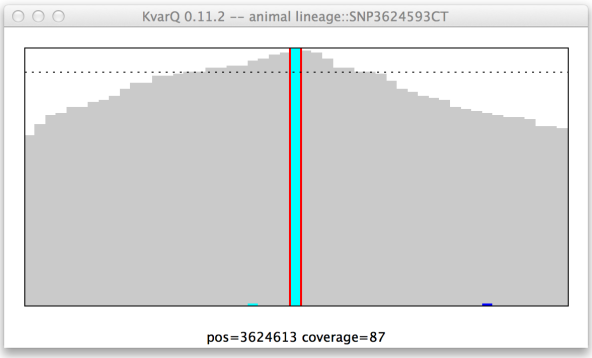


Figure 1.6: Coverage of a single nucleotide polyphormism (SNP), **wildtype genotype**.
Note: There's also the (quite unlikely) possibility that there is a new (i.e. not-looked-for) SNP. This example shows the `SNP3304966GA`. The bottom display (`coverage=91 mutations=91x G`) makes it clear that we have indeed the wild type.

1.4.1 Rolling your own testsuite

KvarQ makes it very simple to write new testsuites. It is probably easiest to take a pre-existing testsuite and adapt it to your needs. All testsuites shipped with KvarQ are well annotated and there are some articles in the *tutorial section* that show how to adapt the testsuites in the `testsuites/example/` directory.

1.4.2 Versions, Compatibility

The following problems can arise when different versions of testsuites are used

- A testsuite is not compatible with the KvarQ version that loads it. To avoid this scenario, the module global `GENES_COMPATIBILITY` is compared with the module global `kvarq.genes.COMPATIBILITY` version of KvarQ running it. The first number must be matched exactly and the second number must be equal *or smaller* to the one defined in the genes package. Whenever KvarQ introduces new features that break the backwards-compatibility with the testsuites, the first number is increased.
- A testsuite is loaded to display data from a `.json` file that was generated by testsuite with a different number. The moduel global `VERSION` tells the version of the testsuite defining it. Upon a backwards-compatible change (e.g. deletion of a previous test), the minor number is increased by one. Note that introductions of new tests are **not** backwards compatible because the new version of the testsuite will be looking for non-existing tests when loading data generated with an old version.

1.4.3 Annotated example

The following is a dump of the extensively annotated testsuite `testsuites/examples/example.py` included with KvarQ

```
# this is an example testsuite that illustrates how to write simple
# SNP/region based testsuite to be used with kvarq

# the testsuite can be included during the scanning by using the
# command line parameter '-t' or in the configuration window in the GUI

# see the kvarq documentation for more information:
# http://kvarq.readthedocs.org/en/latest/testsuites.html
```

```
# the version specifies the version of the testsuite itself; this version
# string is included in the .json scan results
# the minor number should be increased every time the file changes. the major
# number should be increased when the changes are not backwards compatible
# (e.g. when a new test is added)

VERSION = '0.1'

# this version is compared against the COMPATIBILITY global module variable
# defined in kvarq.genes
# as before, compatibility is warranted if the first number is equal and the
# second equal or lower (than the one defined in kvarq.genes)

GENES_COMPATIBILITY = '0.0'


# we use these classes to define our testsuite
from kvarq.genes import Genotype, Gene, Test, Testsuite, Reference, SNP, TemplateFromGenome

# load hypothetical MTB ancestor genome from '../MTBC' directory
# (shipped together with KvarQ)
from kvarq.genes import Genome
import os.path
MTBC_dir = os.path.join(os.path.dirname(__file__), os.pardir, 'MTBC')
ancestor_path = os.path.join(MTBC_dir, 'MTB_ancestor_reference.bases')
ancestor = Genome(ancestor_path, 'MTB ancestor')

# use this for loggging (displayed on console / in main GUI window)
from kvarq.log import lo


# references tell where more information ont he mutations can be found
tbdream = Reference('TBDReamDB : see http://tbdreamdb.com/')

# the first genotype simply signals isoniazid resistance
inhA = Genotype('Isoniazid resistance')
# the second genotype also signals isoniazid resistance but indicates
# the gene to which it belongs to -- this enables output of resistance
# mutation in the familiar gene.XposX format
katG = Genotype('Isoniazid resistance', Gene(ancestor,'katG', 2153889, 2156111, plus_strand=False))

# define two SNPs : 1673432TA and 1673432TC -- only specified mutations will
# be found (i.e. 1673432TATG would not be reported)
# note that the SNP is simply the "template" for that will be used when scanning
# for mutations in the FastQ reads; the "test" as a whole defines a template,
# a genotype (inhA in this case) and the resource from the information is drawn
SNP1 = Test(SNP(genome=ancestor, pos=1673432, orig='T', base='A'), inhA, tbdream)
SNP2 = Test(SNP(genome=ancestor, pos=1673432, orig='T', base='C'), inhA, tbdream)

# define a (short) region that should be scanned for ANY mutations here we're
# interested in the codon 2155167-2155169; by specifying where the gene is read
# from (minus strand) and the position of the amino acid is produced by this
# codon (in this case the gene starts at 2153889, therefore the amino acid is
# ((2155167-2153889)/3 +1)=427) it is later possible to check for (non)
# synonymous mutations as before, the "test" consists of a template, a genotype
# and a resource (but the "template" is a region and not a SNP as before)
katG_codon = Test(TemplateFromGenome(genome=ancestor, start=2155167,
```

```
    stop=2155169, direction='-', aa_pos0=(2155167-2153889)/3 +1), katG,
    tbdream)


# it's important to NAME the testsuite the SAME AS THE FILENAME up to the first
# dash !  (e.g. it's possible to rename this file to "example-0.1.py")
example = Testsuite([SNP1, SNP2, katG_codon], VERSION)


# note that this testsuite is very simple and will simply eport any mutations
# found in the FastQ file -- often it makes sense to subclass the Testsuite
# class (and redefine the _analyse method) to get a fine-grained control on how
# the mutations are synthesized into a result... see the source code of
# kvarq.genes.phylo as an example
```

1.5 Tutorial

The following examples show how KvarQ can be used to quickly analyze genomic data in `.fastq` format. All examples assume that you have *successfully downloaded and installed KvarQ*, but have no other prerequisites. The tutorials are ordered from simpler to more complicated.

1.5.1 Ebola Outbreak 2014

Gire et al have sequenced some 99 virus genomes during the 2014 Ebola outbreak and immediately released all sequence data to "facilitate rapid global research". In the following, we're going to *develop a simple testsuite* that allows KvarQ to say whether a `.fastq` file is from *a)* a ebola virus, *b)* the 2014 outbreak, and *c)* from any of the three sublineages defined in the paper (see figure 4A).

Creating the Testsuite

The following steps led to the Kvarq Ebola sierraleone14 testsuite.

All necessary supplementary materials can be downloaded from the Science webpage. In particular, we're interested in

- Table S2 with the accession numbers of all 99 sequenced genomes. This are serial isolates from 78 patients.
- File S1 that contains `ebov.mafft.fasta` with the sequence alignment. There are 20 sequences before 2014 and 81 sequences from 2014. The sequences form Sierra Leone (sequenced in this paper) are identified by comparing them to the ID column in Table S2.
- Table S4 with the sheet `2014_specific_snps` that lists all SNPs that were found in the new sequences.

With these three files we can generate a list of SNPs that are unique to the isolates from Sierra Leone. Gire et al define three sub-lineages (figure 4A) and by comparing the number of sequences that have specific SNPs we can compile the following table (see the script to extract the SNPs on github).

Position	Ancestral	Derived	Sublineage
800	C	T	SL2
1849	T	C	SL1
6283	C	T	SL1
8928	A	C	SL2
10218	G	A	SL3
13856	A	G	SL1
15660	T	C	SL1
15963	G	A	SL2
17142	T	C	SL2

Creating a testsuite from this data is quite straightforward. First, we choose a reference genome to extract data from. This reference genome should have the ancestral genotype for every SNP that we define. Because we are only interested in SNPs from the 2014 strains, we simply take a genome from a previous isolate, for example the sequence `EBOV_1976_KC242801` from the file `ebov.mafft.fasta` and we save it in a new file called `EBOV76.fasta`. this file is then loaded as a `Genome` in the testsuite:

```
# old ebola genome from previous outbreak
EBOV76 = Genome(os.path.join(os.path.dirname(__file__), 'EBOV76.fasta'))
```

Next, we define the a `Reference` that identifies the source of the data. Then, we `Genotype` can be bound to a gene, but in our case we simply specify it by name.

```
gire14 = Reference('Gire et al (2014) doi 10.1126/science.1259657')

# sub-lineages as defined in gire14
SL1 = Genotype('SL1')
SL2 = Genotype('SL2')
SL3 = Genotype('SL3')
```

In the next step we define the actual `SNPs` and bind them to the genotypes defined above.

```
# SNPs extracted from primary data using suppl/_extract_SNPs.py
SNPs = [
        Test(SNP(genome=EBOV76, pos=800, orig='C', base='T'), SL2, gire14),
        Test(SNP(genome=EBOV76, pos=1849, orig='T', base='C'), SL1, gire14),
        Test(SNP(genome=EBOV76, pos=6283, orig='C', base='T'), SL1, gire14),
        Test(SNP(genome=EBOV76, pos=8928, orig='A', base='C'), SL2, gire14),
        Test(SNP(genome=EBOV76, pos=10218, orig='G', base='A'), SL3, gire14),
        Test(SNP(genome=EBOV76, pos=13856, orig='A', base='G'), SL1, gire14),
        Test(SNP(genome=EBOV76, pos=15660, orig='T', base='C'), SL1, gire14),
        Test(SNP(genome=EBOV76, pos=15963, orig='G', base='A'), SL2, gire14),
        Test(SNP(genome=EBOV76, pos=17142, orig='T', base='C'), SL2, gire14),
    ]
```

Finally, we define a new `Testsuite` from these SNPs and instantiate it to a variable called `sierraleone14`, which must be the same name as python file. We could use the standard testsuite:

```
sierraleone14 = Testsuite(SNPs, VERSION)
```

But we instead choose to define a new testsuite called `CountGenotype` that subclasses the `_analyse` method, to summarize all SNPs into one line that shows the genotype and the number of SNPs found for this genotype. See the complete testsuite on github.

Creating a testsuite from this test data is quite straightforward: simply define each of the `SNPs` as a `Test` and instantiate a `Testsuite`. As a bonus, the Kvarq Ebola sierraleone14 testsuite overrides the `_analyse` method of the testsuite to display how many of the specified SNPs have been found for every sublineage. The reference genome `EBOV76.fasta` is the first genome found in the file `ebov.mafft.fasta`.

View the complete source code of the testsuite on github.

Running the Testsuite

Choose any of the patients, e.g. `EM119` from table S2. The corresponding accession number KM233042 in the nucleotide archive yields another link into the biosample database : SAMN02951962, from where the raw sequencing data can be downloaded. NCBI stores the `.fastq` file in `.sra` format, but this can easily be converted after download using the `fastq-dump` command from the SRA Toolkit.

Now simply download the Kvarq Ebola sierraleone14 testsuite, start the *KvarQ GUI*, *load the testsuite* and analyze the `.fastq` file, or launch KvarQ from the *command line*. The resulting `.json` file can be opened in the *explorer* and should show that the sample from `EM119` is sublineage three, showing all the 6 SNPs from SL1, the 4 SNPs from SL2, and the SNP from SL3.

Downloading the sample from `EM120` (biosample SAMN02951963) and analyzing it the same way shows that this sample also is positive for the 6 SNPs from SL1, and the 4 SNPs from SL2, but that is missing the SNP from SL3 (opening the SNP with the explorer shows that it has the original base `G` at position 10218).

1.5.2 Creating a new SNP testsuite

After reading the interesting article A robust SNP barcode for typing Mycobacterium tuberculosis complex strains I thought it would be nice to analyze some `.fastq` files with that new barcoding scheme.

To get things done quickly, I was browsing through the testsuites in the `testsuites/examples` directory and found a testsuite called `SNPs.py` that looked promising. This testsuite defines a function that loads SNP declarations from a `.tsv` file that can easily be edited with a popular spreadsheet program.

```
here = os.path.dirname(__file__)
# we borrow the reference from the ../MTBC testsuites
genome_path = os.path.join(here, os.path.pardir, 'MTBC', 'MTB_ancestor_reference.bases')
genome = Genome(genome_path, 'MTB ancestor')
ref = Reference('specify reference here')
# load SNP information from .tsv file (can be edited with Excel)
SNPs = tsv2SNPs(os.path.join(here, 'SNPs.tsv'), genome, ref)
```

The format of the `.tsv` file is straightfoward:

lineage1	3920109	G/T
lineage1	3597682	C/T
lineage1	1590555	C/T
lineage2	1834177	A/C
lineage2	3304966	G/A
...	...	...

There is simply an identifier, followed by the position of the SNP within the reference genome (loaded from the file `../MTBC/MTB_ancestor_reference.bases` in the example), then the original base, and finally the derived base. Actually, the SNPs defined in this example testsuite are the same as the ones used for the main lineage classification in the testsuite `MTBC/phylo`. We can quickly confirm this by performing a *scan* using the `MTBC/phylo` and the `examples/SNPs` testsuites and comparing the result (type these commands in KvarQ's root directory)

```
kvarq scan -l MTBC/phylo -l examples/SNPs tests/fastqs/N0116_1_hits_1k.fastq N0116_phylo_SNPs.json
kvarq illustrate -r N0116_phylo_SNPs.json
```

This should result in the following output:

```
examples/SNPs
-------------
['lineage2::SNP1834177AC', 'lineage2::SNP3304966GA']

MTBC/phylo
```

```
----------
'lineage 2 -- low coverage (median below 10x)'
```

So indeed both testsuites report lineage2 – because `examples/SNPs` does not subclass `kvarq.genes.Testsuite`, the result is simply the list of SNPs that were found in the file, while `MTBC/phylo` fuses the two SNPs into one lineage result and warns at the same time of low coverage, but that's material for another tutorial post...

Coming back the SNP barcoding: it's simple enough to compile a list of all SNPs mentioned in the paper. It starts like this:

lineage1	615938	G/A
lineage1.1	4404247	G/A
lineage1.1.1	3021283	G/A
lineage1.1.1.1	3216553	G/A
lineage1.1.2	2622402	G/A
lineage1.1.3	1491275	G/A
lineage1.2.1	3479545	C/A
...	...	...

So let's first create a new directory for the testsuite-to-be-created, calling it `testsuites/MTBC-SNP-barcodes`. Then we copy the following files

- `testsuites/MTBC-SNP-barcodes/coll14.py` : a copy of the file `testsuites/examples/SNPs.py`, will be modified below
- `testsuites/MTBC-SNP-barcodes/coll14.tsv` : the SNP list extracted from the paper; you can download the list from github

Some parts of the example testsuite have to be modified accordingly

```
1  VERSION = '0.1'
2  GENES_COMPATIBILITY = '0.0'
3
4  import os.path
5
6  from kvarq.genes import Genome, Reference, SNP, Test, Testsuite, Genotype
7
8  def tsv2SNPs(path, genome, reference):
9
10     tests = []
11     for line in file(path):
12
13         parts = line.strip().split('\t')
14         name = parts[0]
15         pos = int(parts[1])
16         bases = parts[2].split('/')
17
18         snp = SNP(genome=genome, pos=pos, orig=bases[0], base=bases[1])
19         test = Test(snp, Genotype(name), reference)
20         tests.append(test)
21
22     return tests
23
24 here = os.path.dirname(__file__)
25 genome = Genome(os.path.join(here, 'MTB_ancestor_reference_coll.bases'), 'MTB ancestor')
26 coll14 = Reference('Coll et al (2014) -- doi: 10.1038/ncomms5812')
27 SNPs = tsv2SNPs(os.path.join(here, 'coll14.tsv'), genome, coll14)
28
29 coll14 = Testsuite(SNPs, VERSION)
```

Remarks

- line 1 : it doesn't really matter what `VERSION` we specify, but it's important to increase it when the testsuite is modified to *maintain compatibility*
- line 25 : because the reference genome `MTBC/MTB_ancestor_reference.bases` that was used in the `MTBC/phylo` testsuite has already the derived base in some of the SNPs defined in `coll14.tsv`, we cannot use it as a reference genome (KvarQ asserts that the reference genome has the ancestral base for all defined SNPs to prevent errors). therefore, I have assembled a new reference genome that has the ancestral base for all SNPs
- line 26 : the reference for the testsuite is the original publication from which the SNPs are taken
- line 27 : the SNPs are read rom the file `coll14.tsv`
- line 29 : *the testsuite must be named like the file*

Now let's see whether KvarQ accepts the new testsuite: the command `kvarq info -l MTBC-SNP-barcodes/coll14` should produce the following output:

```
version=0.12.2
testsuites=MTBC-SNP-barcodes/coll14-0.1[62:3162bp]
sum=62 tests,3162bp
sys.prefix=/Library/Frameworks/Python.framework/Versions/2.7
```

So the new testsuite is accepted and KvarQ tells us that it contains 62 tests totaling 3162 base pairs (that's 62 times 1 base plus two flanks of 25 base pairs each).

Running the testsuite

Let's first scan a single `.fastq` to make sure the testsuite works as expected. For example from the internet: MTB_98_1833. Then we scan this file with our new testsuite:

```
kvarq scan -p -l MTBC-SNP-barcodes/coll14 MTB_98_1833.fastq.gz MTB_98_1833.json
```

After a couple of minutes we can examine the result of the scan:

```
kvarq illustrate MTB_98_1833.json
kvarq explorer MTB_98_1833.json
```

This shows us the file contained at the same time SNPs characteristic for lineage 2 and lineage 4, and that the reads are quite short (around 35 base pairs after quality trimming). Practicaly all SNPs were found (with a coverage ranging from 20 to 50), most in their ancestral variant.

Ok, so everything seems to work and we can proceed scanning our local library of `.fastq` files, by writing a simple bash script

```
#!/bin/bash
mkdir results/
for fastq in /genomes/fastqs/*.fastq; do
    json=`basename "$fastq"`.json
    kvarq -l coll14_scan.log scan -l MTBC-SNP-barcodes/coll14 $fastq results/$json
done
```

Some hours later we have scanned for the SNP barcodes of hundreds of genomes, with a copy of the KvarQ log in the file `coll14_scan.log` and a new `.json` file in the `results/` directory for every genome scanned. This information can then be further analyzed using a script that shows all information in tabular form and can also be downloaded from github (note that the script starts with an underscore _ because it is not a testsuite itself and should not be *auto-discovered*).

The finished testsuite can also be found on github.

1.6 Understanding the Scanning Process

1.6.1 Overview

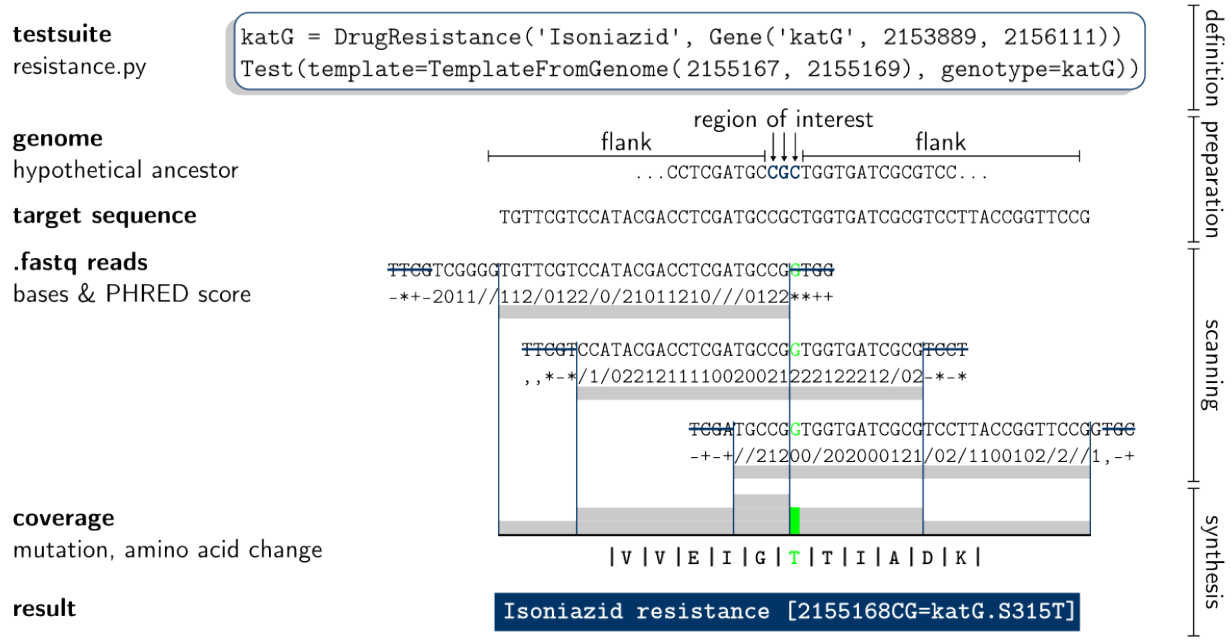


Figure 1.7: Simplified overview of scanning process and preparation.

Testsuites are python source files that define the SNPs of interest (or in this case a 3 base pair long region) as well as other relevant genetic information (in this case the `katG` gene which can confer Isoniazid resistance). This information is used to extract a "target `sequence`" from a hypothetical ancestral MTBC genome: on both sides, additional bases ("flanks") are concatenated to avoid border effects within the sequence of interest. During the scanning process, every read is trimmed depending on its PHRED score (in this case, a quality cutoff of `Q=13` was defined which corresponds to the ASCII character `/`). After the scanning, all reads that matched the target sequence are assembled to a "`coverage`" that indicates the overall coverage depth as well as all detected mutations (green). In a further step, additional information is generated from this coverage (such as the resulting amino acid sequence) and finally a short "result" string is generated that summarizes the result of the scanning process.

The *testsuites* provide a list of **sequences**. The C extension `kvarq.engine` then scans the `.fastq` file and finds all occurrences of these sequences. The list of these occurrences (the **hit list**) is then passed to the module `kvarq.analyser` that maps the reads onto the original sequences and creates **coverages** (see `Coverage`. This coverage is passed back to the different testsuites that calculates the final **results**. The coverages are saved along with the results (and optionally the hit list) into the `.json` file.

Testsuites 1/2

Depending on what *testsuites* are loaded (via the *command line* or the *settings dialog*), KvarQ performs different analysis suitable to detect different genomic markers in different organisms. The testsuite defines a `template` that is then used to generate a `sequence` which is passed along.

Engine

The (C extension) module `kvarq.engine` is the workhorse of the scanning process. It creates multiple threads that scan through the `.fastq` file and returns a list of `kvarq.engine.Hit` that describe the position and overlap of reads from the fastq with the different sequences.

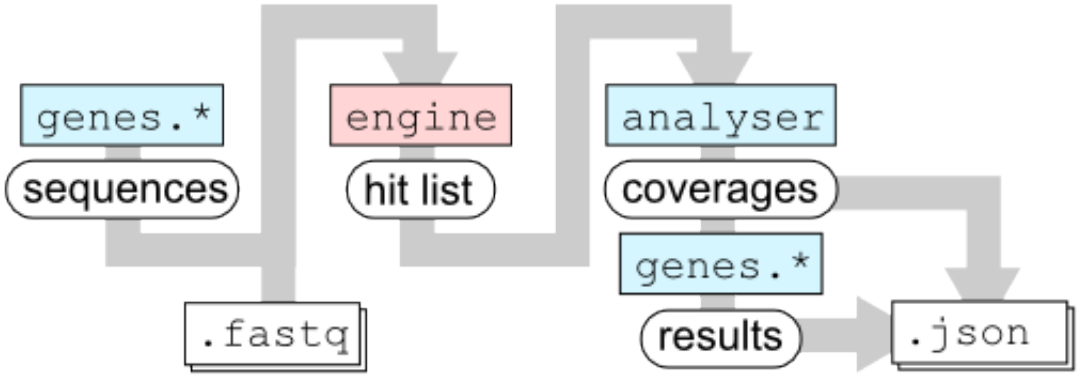


Figure 1.8: This figure illustrates the flow of data inside kvarq.
Python modules/packages are colored in blue, C extensions in red. Rectangles with rounded corners represent data structures passed along the way.

This module is actually called from within *kvarq.analyser* and runs in a separate python thread. It provides some functions that can be called asynchronously from the main (CLI/GUI) thread to monitor the scanning process.

Analyser

The module `kvarq.analyser` takes the hit list from the `kvarq.engine` and applies the overlaps of the reads with the templates, creating a `kvarq.analyser.Coverage` object for every target sequence.

Testsuites 2/2

The coverages are then distributed to the different testsuites and every testsuite does some specific analysis and then reports the final results. For example, the MTBC resistance testsuite (`testsuites/MTBC/resistance.py`) first finds mutations and then determines whether these mutations are synonymous or non-synonymous and outputs the base mutation as well as the resulting change in amino acid if the mutation is non-synonymous.

These final results generated by the testsuites are then saved, along with the coverages, in the `.json` file. The file also contains all relevant scanning parameters (including testsuites and their versions).

1.6.2 Configuration Parameters

The function `kvarq.engine.config()` accepts the following parameters

- `Amin` : ASCII character of the Phred score that corresponds to the minimal quality score of a base calling to be accepted. Use method `kvarq.fastq.Fastq.Q2A()` to translate a Phred score into an ASCII value.
- `minreadlength` : After cutting the individual reads using the provided `Amin`, reads shorter than `minreadlength` are discarded.
- `minoverlap` : Reads that overlap (at the beginning or the end of the sequence) with less bases than the specified values are not reported.
- `maxerrors` : Reads that differ in more than `maxerrors` base positions are not considered for a match.
- `nthreads` : Number of threads to use in parallel for scanning the `.fastq` file.

These parameters can be set using *command line switches* or in the *settings dialog*.

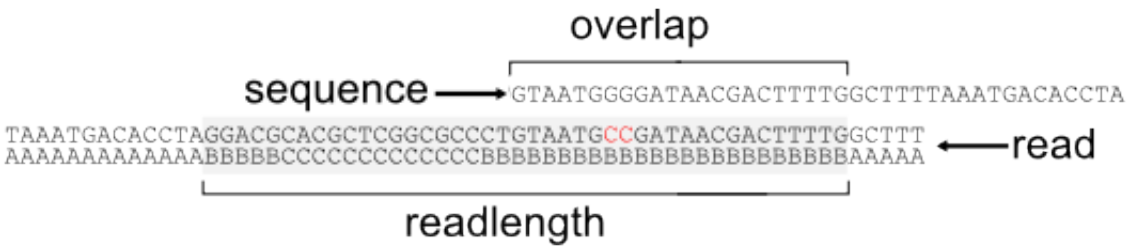


Figure 1.9: This figure illustrates the different configuration parameters for `kvarq.engine`
In this example, `Amin='B'` causes that only the gray part of the read (number of bases in this part is **readlength**) is considered when the read is aligned to the different sequences. The **overlap** is the number of bases that the read and the sequence have in common. In this example the read is aligned despite of the two bases that differ from the sequence – this is only the case if `maxerrors>=2`.

1.7 Hacking KvarQ

This chapter is a good starting point if you intend to change anything in the KvarQ source code, especially in the modules `kvarq.genes` and `kvarq.analyse`. If you're merely interested in writing a new *testuite*, you might also find some valuable information here in addition to the example testsuites shipped with KvarQ. Make sure to have read the *overview* first.

1.7.1 About flanks

In general, KvarQ searches for a given sequence within the genome and checks for mutations within that "region of interest". Because `.fastq` reads are only accepted if the overlap between the target sequence and the read has a specified minimum overlap (*minreadlength*), the coverage rapidly falls on both extremities of the target sequence. For this reason, a region of interest is flanked with supplementary bases on either side. Mutations in these flanks are disregarded, as their sole purpose is to avoid to have the border effect with low coverage within the region of interest. `TemplateFromGenome` read their bases from a `Genome` and can easily add an arbitrary flank on either side of the region of interest.

1.7.2 About positions

- All positions within a `Genome` are relative to the index **one**. I.e. the first codon of the genome (if it were coding) would be the bases 1-3 which correspond to the first three bytes in the file (at file positions 0-2). Positions increase in the reading direction of the plus strand.
- The `TemplateFromGenome`'s `start` and `stop` attributes refer to positions in the genome.
- `Sequences` are simply strings of bases and therefore, indexing within sequences starts at the first character of this sequence string. This sequence string may contain flanks on both sides. The (optional) attribute `start` refers to the first base **after** the flank (i.e. the first base of the sequence of interest).
- In a `Coverage`, positions simply refer to its `Sequences` on the plus strand. A coverage has no `pos` attribute and the `start`, `stop` attributes correspond to the (plus strand) sequence's `left` and `right`.
- The `Hit` structure

1.7.3 About the complementary DNA strand

In general, everything refers to the + strand of the genome. Just before scanning, the analyser creates a complementary copy of every `Sequence` in `Analyser.scan()`. When assembling the hits in `Coverage.apply_hit()`, hits on the complementary strand are mapped on the positive strand again using the methods `Sequence.plus_idx()` and `Sequence.plus_base()`. Finally, the `TemplateFromGenome` and the `Gene` attached to a `Test` know whether the + or the complementary − strand is coding and accordingly converts the base mutations in (non-) synonymous amino acid changes.

1.7.4 Sequence of tests

The `Analyser` is initialized with a set of *testsuites* that define each a given number of `Test`. Just before scanning, the analyser creates a list (actually a `OrderedDict`) that contains every test of every testsuite. The base sequences the `engine` searches in the `.fastq` version is ordered in the same sequence (with the complementary base sequences added at the end of the list). Later on, the `Coverage` are ordered in the same sequence and everything that gets saved to the `.json` file (in `encode()`) uses the same sequence again. To be able to reconstruct the same order when loading a `.json` file, the `decode()` tries to identify every test by its name (as returned by its `__str__()` method) and reconstructs the same sequence of tests. The `Analyser[...]` method retunrs a `Coverage` and accepts integers as well as strings and `Test`).

1.7.5 About clonal variants ("heterozygous calls")

When the DNA is not extracted with great care from single colonies, this can easily result in mixed DNA from different clones. Although KvarQ is not designed to interpret polyclonal variants, the `kvarq.analyse.Coverage` lists detailed information about the frequency of every mutation and new testsuites can use this information to interpret the results of mixed colony sequencing.

Currently, the *explorer* displays clonal variants (defined as base calls with the most dominant base below 90%) with ~ sign, and the `MTBC/phylo` as well as the `MTBC/resistance` testsuites display a remark, when sequencing date seems to stem from a mixed population sequencing.

1.8 Glossary

FastQ file read about FastQ files on Wikipedia

Phred (quality) score read bout quality scores on Wikipedia

MTBC *Mycobacterium tuberculosis* complex

1.9 How to cite

If you use KvarQ for your scientific research, please use the following BibTeX entry (also available in `nbib format`)

```
@article{steiner14kvarq,
  title={KvarQ: targeted and direct variant calling from fastq reads of bacterial genomes},
  author={Steiner, A and Stucki, D and Coscolla, M and Borrell, S and Gagneux S},
  journal={BMC Genomics},
  volume={15},
  pages={881},
  year={2014},
  doi={doi:10.1186/1471-2164-15-881},
```

```
  url={http://www.biomedcentral.com/1471-2164/15/881}
}
```

1.10 Changelog

1.10.1 version 0.12.2

- new `examples/` testsuites and *tutorials*, leading to development of new testsuites ebola-sierraleone14 and MTBC-SNP-barcodes
- new *testsuite discovering/loading* mechanism
- easier selection of testsuites via *GUI*
- better support for *clonal variants*
- `MTBC/resistance` scans `pncA` gene; mutations in `rrsS` and `rrsK` (ribosomal RNA) are labeled correctly
- *summarize* command
- added support for more `.fastq.gz` formats, including paired files
- increased test coverage

1.10.2 version 0.12.1

- support `.fastq.gz` files
- added new ways to *load testsuites*
- some bugfixes in `scripts/table_{scan|combine}.py`
- moved additional *testsuites* into their own repositories

1.10.3 version 0.11.3

- (first public version, pushed to github)
- compile on windows without having to install pthread files
- moved all testsuites into separate `testsuite/` directory
- introduced *testsuite compatability checks*
- updated `MTBC.resistance` testsuite (added `katG.279`, `rpoC.N698H`, renamed `rrsK`)

1.10.4 version 0.11.2

- renamed to KvarQ (previous name was “pyseq”)
- made xlrd, xlwt optional dependencies (import/export data as .csv)
- polished GUI somewhat

1.10.5 version 0.11.1

- internally use `kvarq.analyse.Coverage` instead of `kvarq.genes.Test` to identify sequences and hits
- more compact file format
- support legacy `.json` files; legacy testsuites can also be loaded separately (some are included in `testsuites/legacy/`)

1.10.6 version 0.10.10

- enabled multiple use of same `kvarq.genes.Test` for different `kvarq.genes.Testsuite`

1.10.7 version 0.10.9

- implemented all plotting in `Tkinter` – `matplotlib` not needed anymore
- added settings dialog to GUI
- moved non-published tests into separate files in `testsuites/` directory

1.10.8 version 0.10.8

- added new fields to stats : `nseqhits`, `records_parsed`
- added new attributes to `kvarq.fastq.Fastq` : `readlength`, `records_approx`
- (these fields are also displayed in the json explorer)
- added terminal color support
- improved FastQ quality score decoding
- improved `scripts/table_combine.py` (insert data into existing table)
- include html documentation in distributions
- testsuites can be loaded from arbitrary locations (see *Rolling your own testsuite*)

1.10.9 version 0.10.7

- relaxed `.fastq` file format specifications

1.10.10 version 0.10.6

- `.fastq` file format checking (both in kvarq.fastq and kvarq.engine)
- give every thread 10 file junks to keep scanning speed constant until the end

1.10.11 version 0.10.5

- "batch processing" in `kvarq.gui.{simple|explorer}`
- resistances output aa number
- cleaned up output spoligo/resistances
- do not include hits in .jsons when using gui/simple