vtam Documentation

Release 0.2.0

Aitor Gonzalez, Thomas Dechatre, Reda Mekdad, Emese Meglecz

May 12, 2022

Contents

1 Table of Contents **3**
1.1 Overview of VTAM . . . 3
1.2 Installation . . . 6
1.3 Tutorial . . . 7
1.4 Reference . . . 26
1.5 Input/Output Files . . . 42
1.6 Glossary . . . 47
1.7 List of References . . . 50
1.8 Citation . . . 50
1.9 Change Log . . . 50
1.10 Contributor Guide . . . 53

2 Indices and tables **55**

VTAM is a metabarcoding package with various commands to process high throughput sequencing (HTS) data of amplicons of one or several metabarcoding markers in FASTQ format and produce a table of amplicon sequence variants (ASVs) assigned to taxonomic groups. If you use VTAM in scientific works, **please cite the following article**:

González, A., Dubut, V., Corse, E., Mekdad, R., Dechartre, T. and Meglécz, E.. *VTAM: A robust pipeline for processing metabarcoding data using internal controls*. Submitted to **Methods in Ecology and Evolution**.

Commands for a quick installation:

```
conda create --name vtam python=3.7 -y
python3 -m pip install --upgrade cutadapt
conda install -c bioconda blast
conda install -c bioconda vsearch
python3 -m pip install --upgrade vtam
```

Commands for a quick working example:

```
vtam example
cd example
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_mfzr.yml --until asvtable_taxa
```

The table of amplicon sequence variants (ASV) is here:

```
(vtam) user@host:~/vtam/example$ head -n4 asper1/run1_mfzr/asvtable_default_taxa.tsv
run marker  variant sequence_length read_count      tpos1_run1      tnegtag_run1    ␣
↪14ben01 14ben02 clusterid       clustersize     chimera_borderlineltg_tax_id    ltg_
↪tax_name    ltg_rank         identity         blast_db         phylum  class   order  ␣
↪family  genus   species sequence
run1        MFZR    25      181     478     478     0       0       0       25      1␣
↪       False   131567  cellular organisms      no rank 80      coi_blast_db_20200420␣
↪                                                           ␣
↪ACTATACCTTATCTTCGCAGTATTCTCAGGAATGCTAGGAACTGCTTTTAGTGTTCTTATTCGAATGGAACTAACATCTCCAGGTGTACAATACCTACA
run1        MFZR    51      181     165     0       0       0       165     51      1␣
↪       False                                   coi_blast_db_20200420                  ␣
↪ACTATATTTAATTTTTGCTGCAATTTCTGGTGTAGCAGGAACTACGCTTTCATTGTTTATTAGAGCTACATTAGCGACACCAAATTCTGGTGTTTTAGA
run1        MFZR    88      175     640     640     0       0       0       88      1␣
↪       False   1592914 Caenis pusilla  species 100     coi_blast_db_20200420  ␣
↪Arthropoda      Insecta Ephemeroptera   Caenidae        Caenis  Caenis pusilla ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
```

The database of intermediate data is here:

```
 (vtam) user@host:~/vtam/example$ sqlite3 asper1/db.sqlite '.tables'
FilterChimera                     Sample
FilterChimeraBorderline           SampleInformation
FilterCodonStop                   SortedReadFile
```

(continues on next page)

(continued from previous page)

```
FilterIndel                      TaxAssign
FilterLFN                        Variant
FilterLFNreference               VariantReadCount
FilterMinReplicateNumber         wom_Execution
FilterMinReplicateNumber2        wom_FileInputOutputInformation
FilterMinReplicateNumber3        wom_Option
FilterPCRerror                   wom_TableInputOutputInformation
FilterRenkonen                   wom_TableModificationTime
Marker                           wom_ToolWrapper
ReadCountAverageOverReplicates   wom_TypeInputOrOutput
Run
```

CHAPTER 1

Table of Contents

1.1 Overview of VTAM

1.1.1 Aim

VTAM (Validation and Taxonomic Assignation of Metabarcoding Data) is a metabarcoding pipeline. The analyses start from high throughput sequencing (HTS) data of amplicons of one or several metabarcoding *markers* and produce an *Amplicon Sequence Variant* (ASV or variant) table of validated variants assigned to taxonomic groups.

Data curation (filtering) in VTAM addresses known technical artefacts: Sequencing and PCR errors, presence of highly spurious sequences, chimeras, internal or external contamination and dysfunctional PCRs.

One of the advantages of VTAM is the possibility to optimize filtering parameters for each dataset, based on control samples (*Mock sample* and negative control) to obtain results as close as possible to the expectations. The underlying idea is that if using these adjusted parameters for filtering provide clean controls, the real samples are also likely to be correctly filtered. Clean controls retaining all *expected variants* are the basis of comparability among different sequencing runs and studies. Therefore, the presence of negative control samples and at least one *Mock sample* is crucial.

VTAM can also deal with technical or biological replicates. These are not obligatory but strongly recommended to ensure repeatability.

Furthermore, it is also possible to analyse different primer pairs to amplify the same locus in order to increase the taxonomic coverage (*One-Locus-Several-Primers (OLSP)* strategy).

Major steps:

- *Merge* paired-end reads
- *Sort reads* to samples and replicates according to the presence of sequence tags and primers
- Optimize filtering parameters
- *Filter* out sequence artefacts (denoising) and produce an ASV table
- *Assign* ASVs to taxonomic groups.

- *Pool* ASVs from different but overlapping markers

Features of VTAM:

- Control artefacts to resolve ASVs instead of using clustering as a proxy for species
- Filtering steps use parameters derived from the dataset itself. Parameters are based on the content of negative control and mock samples; therefore, they are tailored for each sequencing run and dataset.
- Eliminate inter-sample contamination and *Tag-jump* and sequencing and PCR artefacts
- Include the analysis of replicates to assure repeatability
- Deal with One-Locus-Several-Primers (*OLSP*) data
- Assign taxa based on NCBI nt or custom database

1.1.2 Implementation

VTAM is a command-line application running on linux, MacOS and Windows Subsystem for Linux (WSL; https://docs.microso

- Wopmars (https://wopmars.readthedocs.io/en/latest/)
- ncbi-blast
- vsearch (Rognes et al., 2016)
- cutadapt (Martin, 2011)
- sqlite

The Wopmars workflow management system guarantees repeatability and avoids re-running steps when not necessary. Data is stored in a sqlite database that ensures traceability.

The pipeline composed of six scripts run as subcommands of vtam:

- *vtam merge*: Merges paired-end reads from FASTQ to FASTA files
- *vtam sortreads reads*: Trims and demultiplexes reads based on sequencing tags
- vtam optimize: Finds optimal parameters for filtering
- *vtam filter*: Creates ASVs, filters sequence artefacts and writes ASV tables
- *vtam taxassign*: Assigns ASVs to taxa
- *vtam pool*: Pools the final ASV tables of different overlapping markers into one

There are a few additional subcommands to help users:

- vtam random_seq Creates a random subset of sequences from fastafiles
- vtam make_known_occurrences to create known occurrence file automatically for the optimize step
- *vtam taxonomy*: Creates a taxonomic TSV file for the taxassign
- *vtam vtam coi_blast_db*: Downloads a precomputed custom BLAST database for the cytochrome C oxidase subunit I (COI) marker gene
- *vtam example*: Generates an example dataset for immediate use with VTAM

Although the pipeline can vary in function of the input data format and the experimental design, a typical pipeline is composed of the following steps in this order :

- merge
- sortreads

- filter (with default, low stringency filtering parameters)
- taxassign
- optimize
- filter (with optimized parameters)
- pool
- taxassign

The command vtam filter should be run twice. First, with default, low stringency filtering parameters. This produces an *ASV table* that is still likely to contain some *occurrences* which should be filtered out. Users should identify from this table clearly unexpected occurrences (variants present in negative controls, unexpected variants in mock samples, variants appearing in a sample of incompatible habitat) and expected occurrences in mock samples. Based on these occurrences, **vtam optimize** will suggest the most suitable parameters that keep all expected occurrences but eliminate most unexpected ones. Then, the command **vtam filter** should be run again, with the optimized parameters.

vtam taxassign has a double role: It will assign ASVs in an input TSV file to taxa, and complete the input TSV file with taxonomic information. The lineages of ASV are stored in a sqlite database to avoid re-running the assignment several times for the same sequence. Therefore running vtam taxassign the second or third time (*e.g.* after the **vtam filter** with optimized parameters or after **vtam pool**) will be very quick and its main role will be to complete the input ASV table with taxonomic information.

If using several overlapping markers vtam pool can be run to pool the ASV tables of the different markers. In this step variants identical in their overlapping regions are pooled together. **vtam pool** can also be used to simply produce one single ASV table from several different runs.

1.1.3 Input data structure

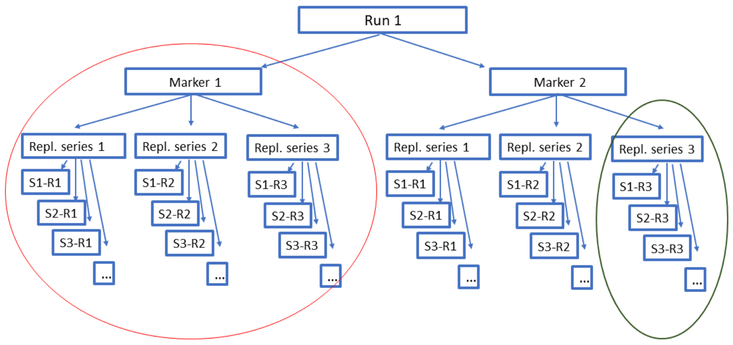


Fig. 1: Figure 1. Input data structure.

The filtering in VTAM is done separately for each run-marker combination. Different runs can be stored in the same database, allowing to pool all the results into the same ASV table.

In case of more than one strongly overlapping *markers*, the results of the same *run(s)* for different markers can also be pooled. Variants identical in their overlapping regions are pooled and presented in the same line of the ASV table.

Replicates are not mandatory, but very strongly recommended to assure repeatability of the results.

Samples belong to 3 categories:

- Mock samples have a known DNA composition. They correspond to an artificial mix of DNA from known organisms.

- Negative controls should not contain any DNA.
- Real samples have an unknown composition. The aim is to determine their composition.

Negative controls and at least one mock sample are required for optimising the filtering parameters.

The mock sample should ideally contain a mix of species covering the taxonomic range of interest and reflect the expected diversity of real samples. It is not essential to have their barcode sequenced in advance if they come from a well-represented taxonomic group in the reference database. In that case, their sequences can be generally easily identified after running the filtering steps and taxonomic assignations with default parameters. However, if there are several species from taxonomic groups weakly represented in the reference database, it is better to barcode the species before adding them to the mock sample. It is preferable to avoid using tissue that can contain non-target DNA (e.g. digestive tract).

Mock samples can contain species that are impossible to find in the real samples (e.g. a butterfly in deep ocean samples). These species are valuable to detect tag jump or inter-sample contaminations in real samples, and thus help to find optimal parameter values of some of the filtering steps. Alternatively, real samples coming from markedly different habitats can also help in the same way.

1.2 Installation

1.2.1 Linux

The easiest way to install and run vtam is via conda (https://docs.conda.io/projects/conda/en/latest/index.html) environment. Download and install Miniconda https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html. Use 3.X python version.

Create a conda environment.

```
conda create --name vtam python=3.7 -y
```

Activate the conda environment.

```
conda activate vtam
```

Install Cutadapt, VSEARCH and BLAST in the environment

```
conda install -c bioconda blast
conda install -c bioconda vsearch
```

Install VTAM via pip

```
pip install vtam
```

You can also verify the BLAST (>= v2.2.26), CutAdapt (>= 2.10) and VSEARCH (>= 2.15.0) versions:

```
blastn -version
cutadapt --version
vsearch --version
```

1.2.2 Singularity

We provide a singularity container in the VTAM github: https://github.com/aitgon/vtam . First you build the image with this command as root:

```
sudo singularity build vtam.sif vtam.singularity
```

Then you can use VTAM directly from the singularity image:

```
singularity run --app vtam vtam.sif --help
singularity run --app vtam vtam.sif merge --help
```

1.2.3 Windows

You can run VTAM on a windows machine using the Windows Subsystem for Linux (WSL)

Install Windows Subsystem for Linux and Ubuntu following these instructions: https://docs.microsoft.com/en-us/windows/wsl/install-win10

Open WSL Ubuntu in a terminal and install conda using Linux instructions: https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html

If it has not been done, you need to install make and gcc:

```
sudo apt-get update
sudo apt install make
sudo apt-get install gcc
conda update -n base -c defaults conda
```

Go on with the VTAM installation as described here *Linux*

Note: You can access your files Windows system from the */mnt* directory of your WSL. For example, execute the following command from the Ubuntu terminal to copy the *file.txt* from *c:tempfile.txt* to your current directory in WSL:

```
cp /mnt/c/temp/file.txt ./file.txt
```

1.2.4 Test your VTAM installation

You can verify the installation of vtam by looking at the VTAM version

```
cd vtam
conda activate vtam
vtam --version
```

The **vtam example** command downloads the example files and create a file structure. A snakemake command will run the whole pipeline. If you get through without an error message your VTAM installation is fully functional.

```
vtam example
cd example
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_mfzr.yml --until asvtable_optimized_taxa
```

1.3 Tutorial

Note: **Important!** Before running any command, do not forget to change directory to vtam and activate the conda environment.

```
cd vtam
conda activate vtam
```

Note: With the exception of BLAST database files and the sqlite database all I/O files of VTAM are text files, that can be opened and edited by a simple text editor (gedit, geany, Notepad++ etc.):

- TSV: Text files with tab separated values. Can also be opened by spreadsheets such as LibreOffice, Excel
- YML: Text files used to provide parameter names and values

1.3.1 Data

In this tutorial, we use a small test dataset based from our previous publication: PMID 28776936. In this dataset, each *sample* was amplified by two overlapping *markers* (mfzr and zfzr), targeting the first 175-181 nucleotides of the COI gene (*Fig. 2*). We had three PCR *replicates* for each sample-marker combination (*Fig. 1*). The samples are *tagged*, so the combination of the forward and reverse tags can be used to identify the origin (sample) of each read.

To reduce run time, the test dataset contains only one *mock sample*, one negative control and two real samples.

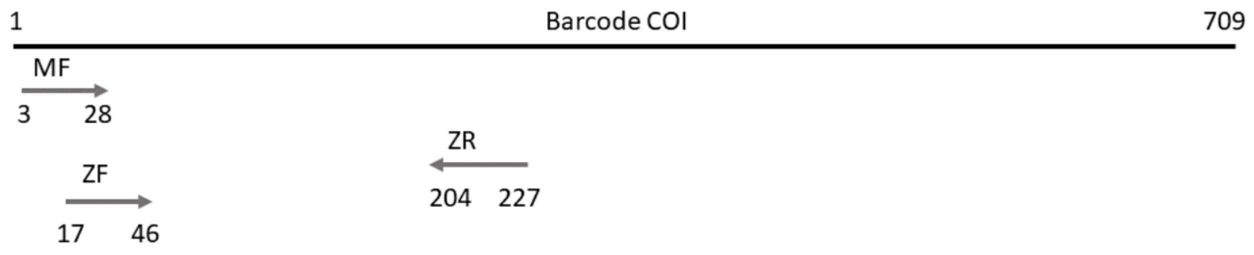


Fig. 2: Figure 2. Positions on the primer on the COI gene used in the test dataset.

You can download these FASTQ files from here with this command:

```
wget -nc https://github.com/aitgon/vtam/releases/latest/download/fastq.tar.gz -O␣
↪fastq.tar.gz
tar zxvf fastq.tar.gz
rm fastq.tar.gz
```

This will create a “FASTQ” directory with 12 FASTQ files:

```
$ ls fastq/
mfzr_1_fw.fastq
mfzr_1_rv.fastq
...
```

mfzr_1_fw.fastq: Forward reads of replicates of the MFZR marker (all samples)

1.3.2 merge: Merge FASTQ files

The simplest use of vtam is to analyze one sequencing run (run1) and one marker (MFZR).

The first step is to *merge* the FASTQ files and transform them into fasta files. It can be skipped, if you have single end reads, or your paired sequences have already been merged and transformed into fasta files.

Create a TSV (tab-separated file), with a header and 10 columns with all the information per FASTQ file pair. We will call it *fastqinfo_mfzr.tsv* in this tutorial and you can download it here: `fastqinfo_mfzr.tsv`. This TSV file will determine, which file pairs should be merged. These files should be all in the *fastq* directory. This directory can contain other files as well, but they will not be analyzed.

The following columns are required in the *fastqinfo_mfzr.tsv*:

- TagFwd
- PrimerFwd
- TagRev
- PrimerRev
- Marker
- Sample
- Replicate
- Run
- FastqFwd
- FastqRev

Tag and primer sequences are in 5' => 3' orientation.

Hereafter are the first lines of the *fastqinfo_mfzr.tsv* file:

```
TagFwd    PrimerFwd    TagRev    PrimerRev    Marker    Sample    Replicate    Run    ␣
↪FastqFwd    FastqRev
tcgatcacgatgt    TCCACTAATCACAARGATATTGGTAC    tgtcgatctacagc    ␣
↪WACTAATCAATTWCCAAATCCTCC    mfzr    tpos1_run1    1    run1    mfzr_1_fw.fastq    ␣
↪mfzr_1_rv.fastq
agatcgtactagct    TCCACTAATCACAARGATATTGGTAC    tgtcgatctacagc    ␣
↪WACTAATCAATTWCCAAATCCTCC    mfzr    tnegtag_run1    1    run1    mfzr_1_fw.fastq    ␣
↪mfzr_1_rv.fastq
```

We propose to work in a project directory called *asper1* (the dataset comes from a project on *Zingel asper*) and copy user created input files such as *fastqinfo_mfzr.tsv* to the *asper1/user_input* directory.

```
asper1
`-- user_input
  `-- fastqinfo_mfzr.tsv
fastq
|-- mfzr_1_fw.fastq
|-- mfzr_1_rv.fastq
|-- ...
```

Run **merge** for all file-pairs in the *fastqinfo_mfzr.tsv*

```
vtam merge --fastqinfo asper1/user_input/fastqinfo_mfzr.tsv --fastqdir fastq --
↪fastainfo asper1/run1_mfzr/fastainfo.tsv --fastadir asper1/run1_mfzr/merged -v --
↪log asper1/vtam.log
```

Note: For info on I/O files see the *Reference section*

This command adds a *merged* directory and a new *fastainfo_mfzr.tsv* file:

```
asper1
|-- run1_mfzr
|   |-- fastainfo.tsv
|   `-- merged
|     |-- mfzr_1_fw.fasta
|     |-- mfzr_2_fw.fasta
|     `-- mfzr_3_fw.fasta
|-- user_input
|   |-- fastqinfo_mfzr.tsv
|-- vtam.err
`-- vtam.log
fastq
|-- mfzr_1_fw.fastq
|-- mfzr_1_rv.fastq
|-- ...
```

The first lines of the *fastainfo_mfzr.tsv* look like this:

```
run     marker     sample     replicate     tagfwd     primerfwd     tagrev     primerrev    ␣
↪mergedfasta
run1     mfzr     tpos1_run1     1     tcgatcacgatgt     TCCACTAATCACAARGATATTGGTAC    ␣
↪tgtcgatctacagc     WACTAATCAATTWCCAAATCCTCC     mfzr_1_fw.fasta
run1     mfzr     tnegtag_run1     1     agatcgtactagct     TCCACTAATCACAARGATATTGGTAC    ␣
↪tgtcgatctacagc     WACTAATCAATTWCCAAATCCTCC     mfzr_1_fw.fasta
```

1.3.3 random_seq: Create a smaller randomized dataset from the main dataset (Optionnal)

The random_seq command is designed to create a smaller randomized dataset with a given number of sequences in each of its output files. This creates a set of files with the number of sequences. Sequences are randomly selected from the fasta files from the given fastadir.

```
vtam random_seq --fastainfo asper1/run1_mfzr/fastainfo.tsv --fastadir asper1/run1_
↪mfzr/merged --random_seqdir asper1/run1_mfzr/randomized --random_seqinfo asper1/
↪run1_mfzr/random_seq_info.tsv --samplesize 50000 -v
```

Note: For info on I/O files see the Reference section

The FASTA files with the randomized reads are written to the *asper1/randomized* directory:

```
asper1
|-- run1_mfzr
|   |-- fastainfo.tsv
|   |-- ...
|   `-- randomized
|     |-- mfzr_1_fw_000_sampled.fasta
|     |-- mfzr_1_fw_001_sampled.fasta
|     |-- ...
|     `-- random_seq_info.tsv
|-- ...
...
```

In addition, the TSV file *asper1/run1_mfzr/sorted/random_seq_info.tsv* lists the information, *i.e.* run, marker, sample and replicate about each randomized FASTA file. The *random_seq_info.tsv* file looks like this:

```
run     marker     sample     replicate     tagfwd     primerfwd     tagrev     primerrev     ␣
↪mergedfasta
run1     mfzr     tpos1_run1     1     tcgatcacgatgt     TCCACTAATCACAARGATATTGGTAC     ␣
↪tgtcgatctacagc     WACTAATCAATTWCCAAATCCTCC     mfzr_1_fw_sampled.fasta
run1     mfzr     tnegtag_run1     1     agatcgtactagct     TCCACTAATCACAARGATATTGGTAC     ␣
↪tgtcgatctacagc     WACTAATCAATTWCCAAATCCTCC     mfzr_1_fw_sampled.fasta
```

1.3.4 sortreads: Demultiplex and trim the reads

There is a single command **sortreads** to *demultiplex* the reads according to *tags* and to *trim* off tags and primers.

The sortreads command is designed to deal with a dual indexing, where forward and reverse tag combinations are used to determine the origin of the reads. This is one of the most complex case of demultiplexing, therefore we implemented **sortreads** to help users.

For simpler cases, we suggest using cutadapt directly, since it is quite straightforward.

```
vtam sortreads --fastainfo asper1/run1_mfzr/fastainfo.tsv --fastadir asper1/run1_mfzr/
↪merged --sorteddir asper1/run1_mfzr/sorted -v --log asper1/vtam.log
```

Note: For info on I/O files see the *Reference section*

The FASTA files with the sorted reads are written to the *asper1/sorted* directory:

```
asper1
|-- run1_mfzr
|   |-- fastainfo.tsv
|   |-- ...
|   `-- sorted
|     |-- run1_MFZR_14ben01_1_mfzr_1_fw_trimmed.fasta
|     |-- run1_MFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
|     |-- ...
|     `-- sortedinfo.tsv
|-- ...
...
```

In addition, the TSV file *asper1/run1_mfzr/sorted/sortedinfo.tsv* lists the information, *i.e.* run, marker, sample and replicate about each sorted FASTA file. The *sortedinfo.tsv* file looks like this:

```
run     marker     sample     replicate     sortedfasta
run1     MFZR     tpos1_run1     1     run1_MFZR_14ben01_1_mfzr_1_fw_trimmed.fasta
run1     MFZR     tnegtag_run1     1     run1_MFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
```

1.3.5 filter: Filter variants and create the ASV table

The **filter** command is typically first run with default parameters. From the output, users should identify clearly unwanted ('*delete*') and clearly necessary ('*keep*') occurrences (see Reference section for details). These false positive and false negative occurrences will be used as input to the **optimize** command. The **optimize** command will then suggest an optimal parameter combination tailored to your dataset. Then **filter** command should be run again with the optimized parameters.

Let's run first the **filter** command with default parameters.

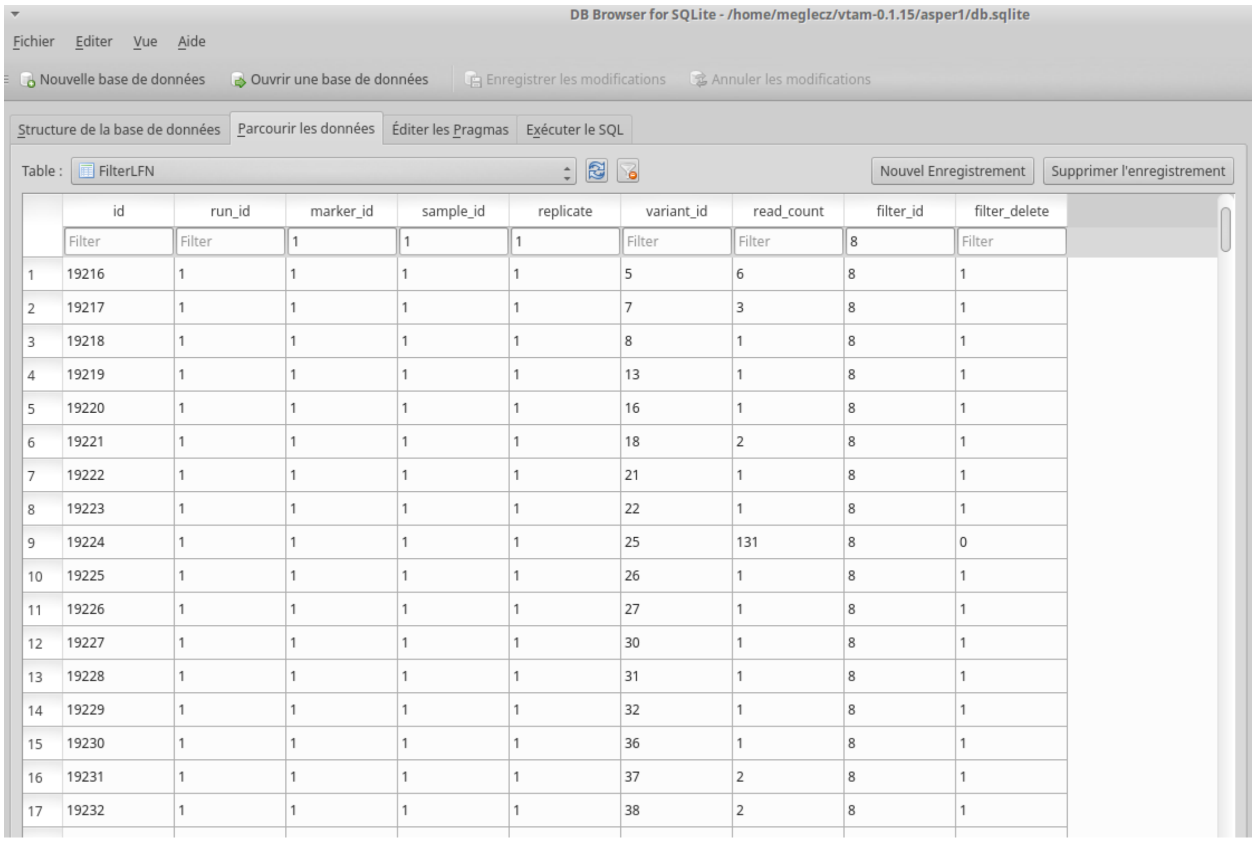
```
vtam filter --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.tsv␣
↪--sorteddir asper1/run1_mfzr/sorted --asvtable asper1/run1_mfzr/asvtable_default.
↪tsv -v --log asper1/vtam.log
```

Note: For info on I/O files see the *Reference section*

This command creates two new files *db.sqlite* and *asvtable_mfzr_default.tsv*:

```
asper1
|-- db.sqlite
|-- run1_mfzr
|   |-- asvtable_default.tsv
|-- ...
...
```

The database *asper1/db.sqlite* contains one table by filter, and in each table occurrences are marked as deleted (filter_delete = 1) or retained (filter_delete = 0). This database can be opened with a sqlite browser program (For example, https://sqlitebrowser.org / or https://sqlitestudio.pl).


DB Browser for SQLite - /home/meglecz/vtam-0.1.15/asper1/db.sqlite
Fichier
Editer
Vue
Aide
Nouvelle base de données
Ouvrir une base de données
Enregistrer les modifications
Annuler les modifications
Structure de la base de données
Parcourir les données
Éditer les Pragmas
Exécuter le SQL
Table : FilterLFN
Nouvel Enregistrement
Supprimer l'enregistrement

	id	run_id	marker_id	sample_id	replicate	variant_id	read_count	filter_id	filter_delete
	Filter	Filter	1	1	1	Filter	Filter	8	Filter
1	19216	1	1	1	1	5	6	8	1
2	19217	1	1	1	1	7	3	8	1
3	19218	1	1	1	1	8	1	8	1
4	19219	1	1	1	1	13	1	8	1
5	19220	1	1	1	1	16	1	8	1
6	19221	1	1	1	1	18	2	8	1
7	19222	1	1	1	1	21	1	8	1
8	19223	1	1	1	1	22	1	8	1
9	19224	1	1	1	1	25	131	8	0
10	19225	1	1	1	1	26	1	8	1
11	19226	1	1	1	1	27	1	8	1
12	19227	1	1	1	1	30	1	8	1
13	19228	1	1	1	1	31	1	8	1
14	19229	1	1	1	1	32	1	8	1
15	19230	1	1	1	1	36	1	8	1
16	19231	1	1	1	1	37	2	8	1
17	19232	1	1	1	1	38	2	8	1

The *asper1/run1_mfzr/asvtable_default.tsv* contains information about the variants that passed all the filters such as the run, maker, read count over all replicates of a sample and the sequence. Hereafter are the first lines of the *asvtable_default.tsv*

```
run     marker     variant     sequence_length     read_count     tpos1_run1     tnegtag_
↪run1     14ben01     14ben02     clusterid     clustersize     chimera_borderline     ␣
↪sequence
run1     MFZR     25     181     478     478     0     0     0     25     1     False     ␣
↪ACTATACCTTATCTTCGCAGTATTCTCAGGAATGCTAGGAACTGCTTTTAGTGTTCTTATTCGAATGGAACTAACATCTCCAGGTGTACAATACCTACA
run1     MFZR     51     181     165     0     0     0     165     51     1     False     ␣
↪ACTATATTTAATTTTTGCTGCAATTTCTGGTGTAGCAGGAACTACGCTTTCATTGTTTATTAGAGCTACATTAGCGACACCAAATTCTGGTGTTTTAGA
```

Note: Filter can be run with the **known_occurrences** argument that will add an additional column for each mock sample flagging expected variants. This helps in creating the **known_occurrences.tsv** input file for the optimization step. For details see the *Reference section*

1.3.6 taxassign: Assign variants of ASV table to taxa

The **taxassign** command assigns ASV sequences in the last column of a TSV file such as the *asvtable_default.tsv* file to taxa.

The **taxassign** command needs a *BLAST database* (containing reference sequences of known taxonomic origin) and the *taxonomy information file*.

A precomputed taxonomy file in TSV format and the BLAST database with COI sequences can be downloaded with these commands:

```
vtam taxonomy -output vtam_db/taxonomy.tsv --precomputed
vtam coi_blast_db --blastdbdir vtam_db/coi_blast_db
```

These commands result in these new files:

```
...
vtam_db
|-- coi_blast_db
|   |-- coi_blast_db_20200420.nhr
|   |-- coi_blast_db_20200420.nin
|   |-- coi_blast_db_20200420.nog
|   |-- coi_blast_db_20200420.nsd
|   |-- coi_blast_db_20200420.nsi
|   └-- coi_blast_db_20200420.nsq
`-- taxonomy.tsv
```

Note: Alternatively, you can use your own custom database or the NCBI nucleotide database *Reference section*

Then, we can carry out the taxonomic assignation of variants in the *asvtable_default.tsv* with the following command:

```
vtam taxassign --db asper1/db.sqlite --asvtable asper1/run1_mfzr/asvtable_default.tsv␣
↪--output asper1/run1_mfzr/asvtable_default_taxa.tsv --taxonomy vtam_db/taxonomy.tsv␣
↪--blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -v --log␣
↪asper1/vtam.log
```

Note: For info on I/O files see the *Reference section*

This results in an additional file:

```
asper1/
|-- run1_mfzr
|   |-- asvtable_default.tsv
|   |-- asvtable_default_taxa.tsv
```

1.3.7 make_known_occurrences: Create file containing the known_occurences.tsv to be used as an inut for optimize

The make_known_occurrences command is designed to automatically create files containing the known and the missing occurences.

```
vtam make_known_occurrences --asvtable asper1/run1_mfzr/asvtable_default.tsv --sample_
↪types asper1/sample_types.tsv --mock_composition asper1/mock_composition.tsv -v
```

Note: For info on I/O files see the Reference section

The TSV files with the known occurrences and the missing occurrences will be written in the asper1 folder

```
asper1
|-- sample_types.tsv
|-- mock_composition.tsv
|-- known_occurrences.tsv
|-- missing_occurrences.tsv
|-- run1_mfzr
|   |-- asvtable_default.tsv
|-- ...
...
```

1.3.8 optimize: Compute optimal filter parameters based on mock and negative samples

The **optimize** command helps users choose optimal parameters for filtering that are specifically adjusted to the dataset. This optimization is based on mock samples and negative controls.

Users should prepare a TSV file (*known_occurrences_mfzr.tsv*) with occurrences to be kept in the results (typically expected variants of the mock samples) and occurrences to be clearly deleted (typically all occurrences in negative controls, and unexpected occurrences in the mock samples). For details see the Reference section.

The example TSV file for the known occurrences of the MFZR marker can be found here : `known_occurrences_mfzr.tsv`.

The first lines of this file look like this:

```
Marker    Run    Sample    Mock    Variant    Action    Sequence
MFZR    run1    tpos1_run1    1        keep  ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
MFZR    run1    tpos1_run1    1        keep  ␣
↪ACTTTATTTTATTTTTGGTGCTTGATCAGGAATAGTAGGAACTTCTTTAAGAATTCTAATTCGAGCTGAATTAGGTCATGCCGGTTCATTAATTGGAGA

...
```

(continues on next page)

(continued from previous page)

```
MFZR    run1    tpos1_run1    1        delete  ␣
↪TTTATATTTCATTTTTGGTGCATGATCAGGTATGGTGGGTACTTCCCTTAGTTTATTAATTCGAGCAGAACTTGGTAATCCTGGTTCTTTGATTGGCG
MFZR    run1    tnegtag_run1    0        delete  ␣
↪TTTATATTTTATTTTTGGAGCCTGAGCTGGAATAGTAGGTACTTCCCTTAGTATACTTATTCGAGCCGAATTAGGACACCCAGGCTCTCTAATTGGAG
```

Note: It is possible to add extra columns with your notes (for example taxon names) to this file after the *Sequence* column. They will be ignored by VTAM.

The **optimize** command is run like this:

```
vtam optimize --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.
↪tsv --sorteddir asper1/run1_mfzr/sorted --known_occurrences asper1/user_input/known_
↪occurrences_mfzr.tsv --outdir asper1/run1_mfzr -v --log asper1/vtam.log
```

Note: For info on I/O files see the Reference section

This command creates four new files:

```
asper1/
|-- db.sqlite
|-- run1_mfzr
|   |-- ...
|   |-- optimize_lfn_sample_replicate.tsv
|   |-- optimize_lfn_read_count_and_lfn_variant.tsv
|   |-- optimize_lfn_variant_specific.tsv
|   |-- optimize_pcr_error.tsv
```

Note: Running vtam optimize will run three underlying scripts:

- **OptimizePCRerror**, to optimize **pcr_error_var_prop**
- **OptimizeLFNsampleReplicate**, to optimize **lfn_sample_replicate_cutoff**
- **OptimizeLFNreadCountAndLFNvariant**, to optimize **lfn_read_count_cutoff** and **lfn_variant_cutoff**.

While **OptimizePCRerror** and **OptimizeLFNsampleReplicate** do not depend on the other two parameters to be optimized, **OptimizeLFNreadCountAndLFNvariant** does. For a finer tuning, it is possible to run the three subscripts one by one, and use the optimized values of **pcr_error_var_prop** and **lfn_sample_replicate_cutoff** instead of their default values, when running **OptimizeLFNreadCountAndLFNvariant**. This procedure can propose less stringent values for **lfn_read_count_cutoff** and **lfn_variant_cutoff**, but still eliminate as many as possible unexpected occurrences, and keep all expected ones.

To run just one subscript, the **–until** flag can be added to the **vtam optimize** command

- until OptimizePCRerror
- unlit OptimizeLFNsampleReplicate
- until OptimizeLFNreadCountAndLFNvariant

e.g.

```
vtam optimize --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.
↪tsv --sorteddir asper1/run1_mfzr/sorted --known_occurrences asper1/user_input/known_
↪occurrences_mfzr.tsv --outdir asper1/run1_mfzr -v --log asper1/vtam.log --until␣
↪OptimizePCRerror

vtam optimize --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.
↪tsv --sorteddir asper1/run1_mfzr/sorted --known_occurrences asper1/user_input/known_
↪occurrences_mfzr.tsv --outdir asper1/run1_mfzr -v --log asper1/vtam.log --until␣
↪OptimizeLFNsampleReplicate
```

Based on the output, create a *params_optimize_mfzr.yml* file that will contain the optimal values suggested for **lfn_sample_replicate_cutoff** and **pcr_error_var_prop**

```
lfn_sample_replicate_cutoff: 0.003
pcr_error_var_prop: 0.1
```

Run **OptimizeLFNreadCountAndLFNvariant** with the optimized parameters for the above two parameters.

```
vtam optimize --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.
↪tsv --sorteddir asper1/run1_mfzr/sorted --known_occurrences asper1/user_input/known_
↪occurrences_mfzr.tsv --outdir asper1/run1_mfzr -v --log asper1/vtam.log --until␣
↪OptimizeLFNreadCountAndLFNvariant --params asper1/user_input/params_optimize_mfzr.
↪yml
```

This step will suggest the following parameter values

```
lfn_variant_cutoff: 0.001
lfn_read_count_cutoff: 20
```

For simplicity, we continue the tutorial with parameters optimized previously, with running all 3 optimize steps in one command.

1.3.9 filter: Create an ASV table with optimal parameters and assign variants to taxa

See the Reference section on how to establish the optimal parameters from the outout of **optimize**. Once the optimal filtering parameters are chosen, rerun the **filter** command using the existing *db.sqlite* database that already has all the variant counts.

Make a *params_mfzr.yml* file that contains the parameter names and values that differ from the default settings.

The *params_mfzr.yml* can be found here: `params_mfzr.yml` and it looks like this:

```
lfn_variant_cutoff: 0.001
lfn_sample_replicate_cutoff: 0.003
lfn_read_count_cutoff: 70
pcr_error_var_prop: 0.1
```

Run filter with optimized parameters:

```
vtam filter --db asper1/db.sqlite --sortedinfo asper1/run1_mfzr/sorted/sortedinfo.tsv␣
↪--sorteddir asper1/run1_mfzr/sorted --params asper1/user_input/params_mfzr.yml --
↪asvtable asper1/run1_mfzr/asvtable_optimized.tsv -v --log asper1/vtam.log
```

Running again **taxassign** will complete the *asvtable_optimized.tsv* with the taxonomic information. It will be very quick since most variants in the table have already gone through the taxonomic assignment, and these assignations are extracted from the *db.sqlite*.

```
vtam taxassign --db asper1/db.sqlite --asvtable asper1/run1_mfzr/asvtable_optimized.
↪tsv --output asper1/run1_mfzr/asvtable_optimized_taxa.tsv --taxonomy vtam_db/
↪taxonomy.tsv --blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -
↪v --log asper1/vtam.log
```

We finished our first analysis with VTAM! The resulting directory structure looks like this:

```
asper1/
|-- db.sqlite
|-- run1_mfzr
|   |-- asvtable_default.tsv
|   |-- asvtable_default_taxa.tsv
|   |-- asvtable_optimized.tsv
|   |-- asvtable_optimized_taxa.tsv
|   |-- fastainfo.tsv
|   |-- merged
|   |   |-- mfzr_1_fw.fasta
|   |   |-- ...
|   |-- optimize_lfn_sample_replicate.tsv
|   |-- optimize_lfn_read_count_and_lfn_variant.tsv
|   |-- optimize_lfn_variant_specific.tsv
|   |-- optimize_pcr_error.tsv
|   `-- sorted
|     |-- run1_MFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
|     |-- ...
|     `-- sortedinfo.tsv
```

1.3.10 Add new run-marker data to the existing database

The same samples can be amplified by different but strongly overlapping markers. In this case, it makes sense to pool all the data into the same database, and produce just one ASV table, with information of both markers. This is the case in our test dataset.

It is also frequent to have different sequencing runs (with one or several markers) that are part of the same study. Feeding them to the same database ensures coherence in variant IDs, and gives the possibility to easily produce one ASV table with all the runs and avoids re-running the **taxassign** on variants that have already been assigned to a taxon.

We assume that you have gone through the basic pipeline in the *previous section*.

Let's see an example on how to complete the previous analyses with the dataset obtained for the same samples but for another marker (ZFZR). The principle is the same if you want to complete the analyses with data from a different sequencing run. First, we need to prepare these user inputs: The directory with the FASTQ files: *fastqinfo_zfzr.tsv*

This is the **merge** command for the new run-marker:

```
vtam merge --fastqinfo asper1/user_input/fastqinfo_zfzr.tsv --fastqdir fastq --
↪fastainfo asper1/run1_zfzr/fastainfo.tsv --fastadir asper1/run1_zfzr/merged -v --
↪log asper1/vtam.log
```

This is the **sortreads** command for the new marker ZFZR:

```
vtam sortreads --fastainfo asper1/run1_zfzr/fastainfo.tsv --fastadir asper1/run1_zfzr/
↪merged --sorteddir asper1/run1_zfzr/sorted -v --log asper1/vtam.log
```

The **filter** command for the new marker ZFZR is the same as in the basic pipeline, but we will complete the previous database *asper1/db.sqlite* with the new variants.

```
vtam filter --db asper1/db.sqlite --sortedinfo asper1/run1_zfzr/sorted/sortedinfo.tsv␣
↪--sorteddir asper1/run1_zfzr/sorted --asvtable asper1/run1_zfzr/asvtable_default.
↪tsv -v --log asper1/vtam.log
```

Next we run the **taxassign** command for the new ASV table *asper1/asvtable_zfzr_default.tsv*:

```
vtam taxassign --db
```

Here, we prepare a new file of known occurrences for the ZFZR marker: *asper1/user_input/known_occurences_zfzr.tsv*. Then we run the **optimize** command with the known occurrences:

```
vtam optimize --db asper1/db.sqlite --sortedinfo asper1/run1_zfzr/sorted/sortedinfo.
↪tsv --sorteddir asper1/run1_zfzr/sorted --known_occurrences asper1/user_input/known_
↪occurrences_zfzr.tsv --outdir asper1/run1_zfzr -v --log asper1/vtam.log
```

At this point, we prepare a new params file for the ZFZR marker: *asper1/user_input/params_zfzr.yml*. Then we run the **filter** command with the optimized parameters:

```
vtam filter --db asper1/db.sqlite --sortedinfo asper1/run1_zfzr/sorted/sortedinfo.tsv␣
↪--sorteddir asper1/run1_zfzr/sorted --params asper1/user_input/params_zfzr.yml --
↪asvtable asper1/run1_zfzr/asvtable_optimized.tsv -v --log asper1/vtam.log
```

Then we run the **taxassign** command of the optimized ASV table:

```
vtam taxassign --db asper1/db.sqlite --asvtable asper1/run1_zfzr/asvtable_optimized.
↪tsv --output asper1/run1_zfzr/asvtable_optimized_taxa.tsv --taxonomy vtam_db/
↪taxonomy.tsv --blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -
↪v --log asper1/vtam.log
```

At this point, we have run the equivalent of the previous section (MFZR marker) for the ZFZR marker. Now we can pool the two markers MFZR and ZFZR. This input TSV file *asper1/user_input/pool_run_marker.tsv* defines the run and marker combinations that must be pooled. The *pool_run_marker.tsv* that looks like this:

```
run     marker
run1     MFZR
run1     ZFZR
```

Then the **pool** command can be used:

```
vtam pool --db asper1/db.sqlite --runmarker asper1/user_input/pool_run_marker.tsv --
↪asvtable asper1/asvtable_pooled_mfzr_zfzr.tsv --log asper1/vtam.log -v
```

Note: For info on I/O files see the *Reference section*.

The output *asvtable_pooled_mfzr_zfzr.tsv* is an asv table that contains all samples of all runs (in this example there is only one run), and all “unique” variants: variants identical in their overlapping regions are pooled into the one line.

Summing read count from different markers does not make sense. In *asvtable_pooled_mfzr_zfzr.tsv* cells contain 1/0 for presence/absence instead of read counts.

Hereafter are the first lines of the *asvtable_pooled_mfzr_zfzr.tsv*.

```
variant_id    pooled_variants    run    marker    tpos1_run1    tnegtag_run1    ␣
↪14ben01    14ben02    clusterid    clustersize    pooled_sequences    sequence
25    25    run1    MFZR    1    0    0    0    25    1    ␣
↪ACTATACCTTATCTTCGCAGTATTCTCAGGAATGCTAGGAACTGCTTTTAGTGTTCTTATTCGAATGGAACTAACATCTCCAGGTGTACAATACCTACA
↪  ␣
↪ACTATACCTTATCTTCGCAGTATTCTCAGGAATGCTAGGAACTGCTTTTAGTGTTCTTATTCGAATGGAACTAACATCTCCAGGTGTACAATACCTACA
137    137    run1    MFZR    1    0    0    0    137    1    ␣
↪ACTTTATTTCATTTTCGGAACATTTGCAGGAGTTGTAGGAACTTTACTTTCATTATTTATTCGTCTTGAATTAGCTTATCCAGGAAATCAATTTTTTT
↪  ␣
↪ACTTTATTTCATTTTCGGAACATTTGCAGGAGTTGTAGGAACTTTACTTTCATTATTTATTCGTCTTGAATTAGCTTATCCAGGAAATCAATTTTTTT
1112    1112,4876    run1    MFZR,ZFZR    1    0    0    0    1112    1    ␣
↪CTTATATTTTATTTTTGGTGCTTGATCAGGGATAGTGGGAACTTCTTTAAGAATTCTTATTCGAGCTGAACTTGGTCATGCGGGATCTTTAATCGGAGA
↪TGCTTGATCAGGGATAGTGGGAACTTCTTTAAGAATTCTTATTCGAGCTGAACTTGGTCATGCGGGATCTTTAATCGGAGACGATCAAATTTACAATG
↪  ␣
↪CTTATATTTTATTTTTGGTGCTTGATCAGGGATAGTGGGAACTTCTTTAAGAATTCTTATTCGAGCTGAACTTGGTCATGCGGGATCTTTAATCGGAGA
```

The *sequence* column is a representative sequence of the *pooled variants*. pooled_sequeces is a list of pooled variants. For details see the *ASV table* format.

Complete the *asvtable_pooled_mfzr_zfzr.tsv* with taxonomic assignments using the **taxassign** command:

```
vtam taxassign --db asper1/db.sqlite --asvtable asper1/asvtable_pooled_mfzr_zfzr.tsv -
↪-output asper1/asvtable_pooled_mfzr_zfzr_taxa.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 --log asper1/
↪vtam.log -v
```

We finished running VTAM for a second marker ZFZR.

The additional data for the ZFZR and the pooled data can be found here:

```
asper1
|-- asvtable_pooled_mfzr_zfzr.tsv
|-- asvtable_pooled_mfzr_zfzr_taxa.tsv
|-- ...
|-- run1_zfzr
|   |-- asvtable_default.tsv
|   |-- asvtable_default_taxa.tsv
|   |-- asvtable_optimized.tsv
|   |-- asvtable_optimized_taxa.tsv
|   |-- ...
...
```

1.3.11 Running VTAM for data with several run-marker combinations

The outcome of some of the filtering steps (**LFNfilter**, **renkonen**) in vtam depends on the composition of the other samples in the dataset. Therefore **vtam is designed to optimize parameters separately for each run-marker combination** and do the filtering steps separately for each of them. However, since run-marker information is taken into account in the vtam scripts, technically it is possible to analyze several dataset (run-marker combination) in a single command.

Let's use the same dataset as before, but run the two markers together. These analyses will give the same results as the previously described pipeline, we will just use fewer commands.

We will define a new output folder *asper2* to clearly separate the results from the previous ones.

For the **merge** command, the format of **fastqinfo** file is as before, but it includes info on both markers.

```
vtam merge --fastqinfo asper2/user_input/fastqinfo.tsv --fastqdir fastq --fastainfo␣
↪asper2/run1/fastainfo.tsv --fastadir asper2/run1/merged -v --log asper2/vtam.log
```

These are the **sortreads** and the **filter** commands:

```
vtam sortreads --fastainfo asper2/run1/fastainfo.tsv --fastadir asper2/run1/merged --
↪sorteddir asper2/run1/sorted -v --log asper2/vtam.log

vtam filter --db asper2/db.sqlite --sortedinfo asper2/run1/sorted/sortedinfo.tsv --
↪sorteddir asper2/run1/sorted --asvtable asper2/run1/asvtable_default.tsv -v --log␣
↪asper2/vtam.log
```

The *asvtable_default.tsv* file contains all variants that passed the filters from both markers. Variants identical in the overlapping regions are NOT pooled at this point, since the optimization will be done separately for each marker-run combination.

This is the **taxassign** command:

```
vtam taxassign --db asper2/db.sqlite --asvtable asper2/run1/asvtable_default.tsv --
↪output asper2/run1/asvtable_default_taxa.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -v --log asper2/
↪vtam.log
```

For the **optimize** command, make one single *known_occurrences.tsv* file with known occurrences for both markers:

```
vtam optimize --db asper2/db.sqlite --sortedinfo asper2/run1/sorted/sortedinfo.tsv --
↪sorteddir asper2/run1/sorted --known_occurrences asper2/user_input/known_
↪occurrences.tsv --outdir asper2/run1 -v --log asper2/vtam.log
```

Each of the output files in the *asper2/run1/optimize* folder will contain information on both markers. The analyses suggest the optimal parameters have been run independently for the two markers. You have to choose the optimal parameters and make *params.yml* files separately for each of them.

Since the optimal parameters for the two markers are likely to be different, you have to run this step separately for the two markers.

The content of the *sortedinfo* is used to define the dataset for which the filtering is done. The *asper2/sorted/sortedinfo.tsv* contains information on both markers. That is why, the filtering by default parameters were run on both markers. In this step, you have to split this file in two. Each of them will contain info on only one marker.

So you will need the following input files:

- *asper2/user_input/params_mfzr.yml*
- *asper2/user_input/params_zfzr.yml*
- *asper2/user_input/readinfo_mfzr.tsv*
- *asper2/user_input/readinfo_zfzr.tsv*

Run **filter** for MFZR:

```
vtam filter --db asper2/db.sqlite --sortedinfo asper2/user_input/sortedinfo_mfzr.tsv -
↪-sorteddir asper2/run1/sorted --asvtable asper2/run1/asvtable_optimized_mfzr.tsv -v␣
↪--log asper2/vtam.log --params asper2/user_input/params_mfzr.yml
```

Run **filter** for ZFZR:

```
vtam filter --db asper2/db.sqlite --sortedinfo asper2/user_input/sortedinfo_zfzr.tsv -
↪-sorteddir asper2/run1/sorted --asvtable asper2/run1/asvtable_optimized_zfzr.tsv -v␣
↪--log asper2/vtam.log --params asper2/user_input/params_zfzr.yml
```

To end this case, we run the **pool** and **taxassign** commands:

```
vtam pool --db asper2/db.sqlite --runmarker asper2/user_input/pool_run_marker.tsv --
↪asvtable asper2/pooled_asvtable_mfzr_zfzr.tsv --log asper2/vtam.log -v

vtam taxassign --db asper2/db.sqlite --asvtable asper2/pooled_asvtable_mfzr_zfzr.tsv -
↪-output asper2/pooled_asvtable_mfzr_zfzr_taxa.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 --log asper2/
↪vtam.log -v
```

We finished running VTAM for the two markers!

1.3.12 Run VTAM with snakemake

Snakemake is a tool to create analysis workflows composed of several steps of VTAM. In this section, we will use a *Snakefile* to run several steps together.

This part of the tutorial supposes that you have read the tutorial on how to *run vtam command by command* for one run-marker combination and you understand the role of each step and the essential input files.

Basic pipeline with snakemake: one run-marker combination

We will work marker by marker. At the root there is a project folder (*asper1*). The analyses related to run-marker will go to different subfolders (eg. *run1_mfzr*) that will contain all related files. We will illustrate the pipeline with the marker MFZR but the same commands can be run later in the same project folder with the marker ZFZR. First make sure that you have the *vtam_db* and *fastq* directories as in the *Data section* of the Tutorial. To setup the pipeline we need the *fastqinfo_mfzr.tsv* file as before and a config file for **snakemake**, called *snakeconfig_mfzr.yml*. We will prepare these files inside a *<project>/user_input* folder as before. The *snakeconfig_mfzr.yml* looks like this:

```
project: 'asper1'
subproject: 'run1_mfzr'
fastqinfo: 'asper1/user_input/fastqinfo_mfzr.tsv'
fastqdir: 'fastq'
known_occurrences: 'asper1/user_input/known_occurrences_mfzr.tsv'
params: 'asper1/user_input/params_mfzr.yml'
blastdbdir: 'vtam_db/coi_blast_db'
blastdbname: 'coi_blast_db_20200420'
taxonomy: 'vtam_db/taxonomy.tsv'
```

Make sure the `snakemake.yml` is in the current working directory. The resulting file tree looks like this:

```
.
|-- asper1
|   `-- user_input
|      |-- fastqinfo_mfzr.tsv
|      `-- snakeconfig_mfzr.yml
|-- fastq
|   |-- mfzr_1_fw.fastq
|   |-- ...
|-- snakefile.yml
```

(continues on next page)

(continued from previous page)

```
`-- vtam_db
  |-- coi_blast_db
  |   |-- coi_blast_db.nhr
  |   |-- ...
  `-- taxonomy.tsv
```

Steps merge, sortreads, filter with default parameters, taxassign

You can run these four steps in one go and create the ASV table with taxonomic assignments with this command:

```
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_mfzr.yml --until asvtable_taxa
```

We find the same directory tree as before:

```
asper1
|-- db.sqlite
|-- run1_mfzr
|   |-- asvtable.tsv
|   |-- asvtable_taxa.tsv
|   |-- fastainfo.tsv
|   |-- merged
|   |   |-- mfzr_1_fw.fasta
|   |   |-- ...
|   `-- sorted
|       |-- run1_MFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
|       |-- ...
|       `-- sortedinfo.tsv
|-- user_input
|   |-- fastqinfo_mfzr.tsv
|   |-- known_occurrences_mfzr.tsv
|   |-- params_mfzr.yml
|   `-- snakeconfig_mfzr.yml
|-- vtam.err
`-- vtam.log
```

The step optimize

You can now create the *asper1/user_input/known_occurrences_mfzr.tsv* based on the informations given by the *asper1/run1_mfzr/asvtable_default_taxa.tsv*.

Then you will run the **optimize** script to look for better parameters for the MFZR marker:

```
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_mfzr.yml --until optimize
```

The resulting optimization files will be found here:

```
asper1
|-- db.sqlite
|-- run1_mfzr
|   |-- ...
|   |-- optimize_lfn_sample_replicate.tsv
|   |-- optimize_lfn_read_count_and_lfn_variant.tsv
```

(continues on next page)

(continued from previous page)

```
|   |-- optimize_lfn_variant_specific.tsv
|   |-- optimize_pcr_error.tsv
```

The steps filter with optimized parameters and taxassign

Define the optimal parameters and create a parameter file: *asper1/user_input/params_mfzr.yml*. Run the filtering and taxassign steps:

```
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_mfzr.yml --until asvtable_optimized_taxa
```

This last command will give you two new ASV tables with optimized parameters:

```
asper1
|-- ...
|-- run1_mfzr
|   |-- ...
|   |-- asvtable_params_taxa.tsv
|   |-- asvtable_params.tsv
```

Add new run-marker data to existing database

The same commands can be run for the second marker ZFZR. You will need the following additional files:

- *asper1/user_input/snakeconfig_zfzr.yml*
- *asper1/user_input/fastqinfo_zfzr.tsv*
- *asper1/user_input/known_occurrences_zfzr.tsv*
- *asper1/user_input/params_zfzr.yml*

The *snakeconfig_zfzr.yml* will look like this:

```
project: 'asper1'
subproject: 'run1_zfzr'
fastqinfo: 'asper1/user_input/fastqinfo_zfzr.tsv'
fastqdir: 'fastq'
known_occurrences: 'asper1/user_input/known_occurrences_zfzr.tsv'
params: 'asper1/user_input/params_zfzr.yml'
blastdbdir: 'vtam_db/coi_blast_db'
blastdbname: 'coi_blast_db_20200420'
taxonomy: 'vtam_db/taxonomy.tsv'
```

Then you can run the same commands as above for the new marker ZFZR:

```
snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_zfzr.yml --until asvtable_taxa

snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_zfzr.yml --until optimize

snakemake --printshellcmds --resources db=1 --snakefile snakefile.yml --cores 4 --
↪configfile asper1/user_input/snakeconfig_zfzr.yml --until asvtable_optimized_taxa
```

These commands will generate a new folder with the same files for the new marker ZFZR. The database *db.sqlite* will be shared by both markers MFZR and ZFZR:

```
asper1
|-- db.sqlite
|-- ...
|-- run1_zfzr
|   |-- asvtable.tsv
|   |-- asvtable_taxa.tsv
|   |-- fastainfo.tsv
|   |-- merged
|   |   |-- zfzr_1_fw.fasta
|   |   |-- ...
|   `-- sorted
|     |-- sortedinfo.tsv
|     |-- run1_ZFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
|     |-- ...
```

The results of the two markers can be pooled as before:

```
vtam pool --db asper1/db.sqlite --runmarker asper1/user_input/pool_run_marker.tsv --
↪asvtable asper1/pooled_asvtable_mfzr_zfzr.tsv --log asper1/vtam.log -v

vtam taxassign --db asper1/db.sqlite --asvtable asper1/pooled_asvtable_mfzr_zfzr.tsv -
↪-output asper1/pooled_asvtable_mfzr_zfzr_taxa.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 --log asper1/
↪vtam.log -v
```

Running snakemake for data with several run-marker combinations

Similarly as before, we can run all run-markers simultaneously. We will use these files:

- *asper2/user_input/snakeconfig.yml*
- *asper2/user_input/fastqinfo.tsv* (info on both markers)
- *asper2/user_input/known_occurrences.tsv* (info on both markers)
- *asper2/user_input/params.yml* (Empty or absent ok)

The *snakeconfig.yml* looks like this:

```
project: 'asper2'
subproject: 'run1'
db: 'db.sqlite'
fastqinfo: 'asper2/user_input/fastqinfo.tsv'
fastqdir: 'fastq'
known_occurrences: 'asper2/user_input/known_occurrences.tsv'
params: 'asper2/user_input/params.yml'
blastdbdir: 'vtam_db/coi_blast_db'
blastdbname: 'coi_blast_db_20200420'
taxonomy: 'vtam_db/taxonomy.tsv'
```

Then you compute the ASV tables and the optimization files with default parameters:

```
snakemake -p --resources db=1 -s snakefile.yml --cores 4 --configfile asper2/user_
↪input/snakeconfig.yml --until asvtable_taxa
```

(continues on next page)

(continued from previous page)

```
snakemake -p --resources db=1 -s snakefile.yml --cores 4 --configfile asper2/user_
↪input/snakeconfig.yml --until optimize
```

Optimized parameter are specific of each marker.

Therefore, it is simpler to run the optimized filter as in the previous section with two files *params_mfzr.yml* and *params_zfzr.yml* for each marker:

- *asper2/user_input/params_mfzr.yml*
- *asper2/user_input/params_zfzr.yml*

To run the **filter** command for each marker, we need to create two *readinfo.tsv* files for each marker based on *asper2/prerun/sorted/readinfo.tsv*:

- *asper2/user_input/sortedinfo_mfzr.tsv*
- *asper2/user_input/sortedinfo_zfzr.tsv*

Then, we can run the filter and taxassign commands with optimized parameters:

```
vtam filter --db asper2/db.sqlite --sortedinfo asper2/user_input/sortedinfo_mfzr.tsv -
↪-sorteddir asper2/run1/sorted --params asper2/user_input/params_mfzr.yml --asvtable␣
↪asper2/run1/asvtable_params_mfzr.tsv -v --log asper2/vtam.log

vtam taxassign --db asper2/db.sqlite --asvtable asper2/run1/asvtable_params_mfzr.tsv -
↪-output asper2/run1/asvtable_params_taxa_mfzr.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -v --log asper2/
↪vtam.log

vtam filter --db asper2/db.sqlite --sortedinfo asper2/user_input/sortedinfo_zfzr.tsv -
↪-sorteddir asper2/run1/sorted --params asper2/user_input/params_zfzr.yml --asvtable␣
↪asper2/run1/asvtable_params_zfzr.tsv -v --log asper2/vtam.log

vtam taxassign --db asper2/db.sqlite --asvtable asper2/run1/asvtable_params_zfzr.tsv -
↪-output asper2/run1/asvtable_params_taxa_zfzr.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 -v --log asper2/
↪vtam.log
```

The resulting directory tree looks like this:

```
asper2
|-- db.sqlite
|-- run1
|   |-- asvtable.tsv
|   |-- asvtable_params_mfzr.tsv
|   |-- asvtable_params_taxa_mfzr.tsv
|   |-- asvtable_params_taxa_zfzr.tsv
|   |-- asvtable_params_zfzr.tsv
|   |-- asvtable_taxa.tsv
|   |-- fastainfo.tsv
|   |-- merged
|   |   |-- mfzr_1_fw.fasta
|   |   |-- ....
|   `-- sorted
|       |-- run1_MFZR_14ben01_2_mfzr_2_fw_trimmed.fasta
|       |-- ...
|       |-- sortedinfo.tsv
|       |-- ...
```

(continues on next page)

(continued from previous page)

```
|-- user_input
|   |-- fastqinfo.tsv
|   |-- known_occurrences.tsv
|   |-- params.yml
|   |-- params_mfzr.yml
|   |-- params_zfzr.yml
|   |-- snakeconfig.yml
|   |-- readinfo_mfzr.tsv
|   `-- readinfo_zfzr.tsv
```

The results of the two markers can be pooled as before:

```
vtam pool --db asper2/db.sqlite --runmarker asper2/user_input/pool_run_marker.tsv --
↪asvtable asper2/pooled_asvtable_mfzr_zfzr.tsv --log asper2/vtam.log -v

vtam taxassign --db asper2/db.sqlite --asvtable asper2/pooled_asvtable_mfzr_zfzr.tsv -
↪-output asper2/pooled_asvtable_mfzr_zfzr_taxa.tsv --taxonomy vtam_db/taxonomy.tsv --
↪blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_20200420 --log asper2/
↪vtam.log -v
```

1.4 Reference

1.4.1 The numerical parameter file

For each vtam command the **params** argument takes an optional YML file that sets parameter values. Omitting the **params** argument will prompt vtam to use default values. The full list of parameters and the default values is shown here:

```
################################################################################
# Parameters of the "merge" command
# These parameters are used by the vsearch --fastq_mergepairs tool
# that underlies the "vtam merge" command
# For a description of these parameters run_name "vsearch --help"
fastq_ascii: 33
fastq_maxee: 1
fastq_maxmergelen: 500
fastq_maxns: 0
fastq_minlen: 50
fastq_minmergelen: 100
fastq_minovlen: 50
fastq_truncqual: 10

################################################################################
# Parameters of the "sortreads" command
# These parameters correspond to the corresponding parametes by cutadapt
# that underlies the "vtam sortreads" command
# For a description of these parameters run_name "cutadapt --help"
cutadapt_error_rate: 0.1 # -e in cutadapt
cutadapt_minimum_length: 50 # -m in cutadapt
cutadapt_maximum_length: 500 # -M in cutadapt

################################################################################
# Parameters of the "filter" command
```

(continues on next page)

(continued from previous page)

```
# This parameter sets the minimal number of reads of a variant in the whole run
global_read_count_cutoff: 2

################################################################################
# Parameters of the "FilterLFN" filter in the "filter" command
# These parameters set the cutoffs for the low frequency noise (LFN) filters
# N_ijk is the number of reads of varinat i, sample j and replicate k
# Occurrence is deleted if N_ijk/N_i < lfn_variant_cutoff
lfn_variant_cutoff: 0.001
# Occurrence is deleted if N_ijk/N_ik < lfn_variant_replicate_cutoff
# This parameter is used if the --lfn_variant_replicate option is set in
# "vtam filter" or "vtam optimize"
lfn_variant_replicate_cutoff: 0.001
# Occurrence is deleted if N_ijk/N_jk < lfn_ sample lfn_sample_replicate_cutoff
lfn_sample_replicate_cutoff: 0.001
# Occurrence is deleted if N_ijk < lfn_ lfn_read_count_cutoff
lfn_read_count_cutoff: 10

################################################################################
# Parameters of the "FilterMinReplicateNumber" filter in the "filter" command
# Occurrences of a variant in a given sample are retained only if it is
# present in at least min_replicate_number replicates of the sample
min_replicate_number: 2

################################################################################
# Parameter of the "FilterPCRerror" filter in the "filter" command
# A given variant 1 is eliminated if N_1j/N_2j < pcr_error_var_prop, where
# variant 2 is identical to variant 1 except a single mismatch
pcr_error_var_prop: 0.1

################################################################################
# Parameter of the "FilterChimera" filter in the "filter" command
# This parameter corresponds to the abskew parameter in the vsearch
# --uchime3_denovo tool that underlies the vtam FilterChimera
# For a description of this parameter run_name "vsearch --help"
uchime3_denovo_abskew: 16.0

################################################################################
# Parameter of the "FilterRenkonen" filter in the "filter" command
# Quantile renkonen distance to drop more extreme values
# For. a 0.9 value will set the 9th decile of all renkonen distances as cutoff
renkonen_distance_quantile: 0.9

################################################################################
# Parameter of the "FilterIndel" filter in the "filter" command
# If 1, skips this filter for non-coding markers
skip_filter_indel: 0

################################################################################
# Parameter of the "FilterCondonStop" filter in the "filter" command
# If 1, skips this filter for non-coding markers
skip_filter_codon_stop: 0
# Translation table number from NCBI [ link]
# Default NCBI translation table 5: stops: ['TAA', 'UAA', 'TAG', 'UAG']
genetic_code: 5

################################################################################
```

(continues on next page)

(continued from previous page)

```
# Parameter of the "MakeAsvTable" filter in the "filter" command
# Cluster identity value to clusterize sequences
cluster_identity: 0.97

################################################################################
# Parameters of the "taxassign" command
# Blast parameter for the minimum query coverage
qcov_hsp_perc: 80
# The LTG must include include_prop percent of the hits
include_prop: 90
# Minimal number of taxa among the hits to assign LTG when %identity
# is below ltg_rule_threshold
min_number_of_taxa: 3
ltg_rule_threshold: 97
```

1.4.2 The command merge

VTAM can start from FASTQ files of paired end metabarcoding data. This command merges paired end sequences using the **fastq_mergepairs** command of vsearch.

A quick introduction to the **vtam merge** command is given in the tutorial *merge: Merge FASTQ files*.

The command line arguments of the **merge** command can be obtained with

```
vtam merge --help
```

The most important arguments are these inputs and outputs:

Inputs:

- *fastqinfo*: TSV file with files to be merged. These files can be compressed in .gz or .bz2.
- **fastqdir**: Path to the directory containing the fastq files

Outputs:

- *fastainfo*: TSV file created by vtam merge. It contains all the info of *fastqinfo* completed by the names of the merged fasta files.
- **fastadir**: Directory to keep the output merged fasta files. If the input files were compressed, the output files will be compressed using the same extension.

The list of numerical parameters can be found in the *The numerical parameter file* section.

1.4.3 The command sortreads

Typically, the sequencing reads contain primers and tags, and this command uses them to attribute each read to a run-marker-sample-replicate.

A quick introduction to the **sortreads** command is given in the tutorial *sortreads: Demultiplex and trim the reads*.

The arguments of the **sortreads** command can be obtained with

```
vtam sortreads --help
```

The most important arguments are these inputs and outputs:

Inputs:

- *fastainfo*: TSV file created by **merge** (allows gzip and bzip2 compressed files). It contains all the info of *fastqinfo* completed by the names of the merged fasta files.
- **fastadir**: Directory containing the merged fasta files.

Outputs:

- **sorteddir**: Directory to keep the output *demultiplexed* fasta files. In this folder, there is also a *sortedinfo* file with the information about each sorted fasta file.

The list of numerical parameters can be found in the *The numerical parameter file* section.

These are the different actions of the sortreads command:

- Sort reads to sample-replicates according to the presence of tags using cutadapt. Exact matches are imposed between reads and tags, and the minimum overlap is the length of the tag.
- Trim reads from primers using cutadapt. Mismatches are allowed (**cutadapt_error_rate**), but no indels. The minimum overlap between the primer and the read is the length of the primer.
- The trimmed sequences are kept if their length is between **cutadapt_minimum_length** and **cutadapt_maximum_length**.

The optionnal argument **no_reverse** can be used if all sequences are correctly oriented and it is not necessary to check the reverse strand. It makes the run faster.

The optionnal argument **tag_to_end** can be used if the tags are located at the edges of the sequences. It make the run faster.

The optionnal argument **primer_to_end** can be used if the primers are located at the edges of the sequences (right after the tags, there is no spacer between them). It makes the run faster.

1.4.4 The command random_seq (OPTIONNAL)

When working with large datasets VTAM can subselect a random set of sequences in order to run with less data to process to reduce the running time and the workload on a users machine.

A quick introduction to the **vtam random_seq** command is given in the tutorial *random_seq: Create a smaller randomized dataset from the main dataset (Optionnal)*.

The command line arguments of the **random_seq** command can be obtained with

```
vtam random_seq --help
```

The most important arguments are these inputs and outputs:

Inputs:

- *fastainfo*: TSV file with files to take the sequences from
- **fastadir**: Path to the directory containing the fasta files
- **samplesize**: number of sequences to be randomly selected

Outputs:

- random_seqinfo: TSV file created by vtam random_seq. It contains all the info of *fastainfo* completed by the names of the fasta files containing the randomly selected sequences.
- **random_seqdir**: directory with randomly selected sequences in FASTA format

1.4.5 The command filter

This command chains several steps of the analyses. It starts from *demultiplexed*, *trimmed* reads, fills the sqlite database with variants and read counts, runs several filtering steps and produces *ASV tables*. Each run-marker combination is treated independently during all filtering steps, even if several run-marker combinations are included in the dataset.

A quick introduction to the **filter** command is given in the tutorial *filter: Filter variants and create the ASV table*.

The arguments of the **filter** command can be obtained with

```
vtam filter --help
```

These arguments are the most important inputs and outputs.

Inputs:

- *sortedinfo*: TSV with the information about each sorted read file (output of sortreads). The content of the file will define the dataset to be used for filtering.
- **sorteddir**: Directory containing the demultiplexed fasta files.

Database:

- *db*: Name of an sqlite db. It is used to store and access information on runs, markers, samples, variants, filtering steps and taxonomic assignations. It is created by **filter** if it does not exist and it is completed if it exists already.

Outputs:

- *asvtable*: TSV file containing variants that passed all the filters, together with read count in the different samples.

The list of numerical parameters can be found in the *The numerical parameter file* section.

In a typical run, **filter** should be run twice: Once with light filtering parameters (default) to do a prefiltering step and produce a user-friendly *ASV table*. Based on this ASV table users will identify the clearly *expected* and *unexpected occurrences* (e.g. in negative controls and mock samples). Using these occurrences the **optimize** command will suggest a parameter setting that keeps all expected occurrences and eliminates most unexpected ones. Then **filter** should be run again with the optimized parameter settings.

FilterLFN

This step intends to eliminate occurrences with low read counts that can be present due to light contaminations, sequencing errors and tag jumps.

Let *N_ijk* be the number of the reads of variant *i*, in sample *j* and replicate *k*.

Each *occurrence* can be characterized by the number of the reads of a variant in a given sample-replicate (*N_ijk*). Low read counts can be due to contamination or artefacts and therefore considered as Low Frequency Noise (LFN).

The following LFN filters can be run on each occurrence, and the occurrence is retained only if it passes all of the activated filters:

- LFN_sample_replicate filter: occurrence is deleted if *N_ijk/N_jk* < lfn_sample_replicate_cutoff
- LFN_read_count filter: occurrence is deleted if *N_ijk* < lfn_read_count_cutoff
- LFN_variant filter: occurrence is deleted if *N_ijk/N_i* < lfn_variant_cutoff
- LFN_variant_replicate filter: occurrence is deleted if *N_ijk/N_ik* < lfn_variant_replicate_cutoff

LFN_sample_replicate and **LFN_read_count** filters intend to eliminate mainly sequencing or PCR artefacts. **LFN_variant** and **LFN_variant_replicate filters** intend to eliminate occurrences that are present due to tag jump or slight inter sample contamination.

The **LFN_variant** and **LFN_variant_replicate** are two alternatives for the same idea and they are mutually exclusive. The **LFN_variant** mode is activated by default. Users can change to **LFN_variant_replicate** mode by using the **lfn_variant_replicate** flag in the command line. These filters eliminate occurrences that have low frequencies compared to the total number of reads of the variant (**LFN_variant**) or variant-replicate (**LFN_variant_replicate**) in the whole run-marker set.

In a typical case, the **lfn_variant_cutoff** and **lfn_variant_replicate_cutoff** are the same for all variants for a given run-marker combination. This use should be preferred. However, occasionally, it can be justified to set individual (variant specific) thresholds to some of the variants. This is done using the **cutoff_specific** parameter that takes as a value a TSV file containing the variant specific threshold values. For variants not specified in the file, the value set by **lfn_variant_cutoff** or **lfn_variant_replicate_cutoff** is used.

FilterMinReplicateNumber

FilterMinReplicateNumber is used to retain occurrences only if they are repeatable.

Within each sample, occurrences of a variant are retained only if it is present in at least **min_replicate_number** replicates.

FilterPCRerror

This step intends to eliminate occurrences due to PCR errors. These errors can be more frequent than sequencing errors.

Within each sample, this filter eliminates variants that have only one mismatch compared to another more frequent variant. The read count proportion of the two variants must be below **pcr_error_var_prop** in order to eliminate the least frequent variant.

FilterChimera

Chimera filtering is done by the **uchime3_denovo** command integrated in vsearch. Chimera checking is done sample by sample. Variants classed as chimeras are eliminated. Those classed as *borderline* in at least one sample are flagged and this information will appear in the final ASV table.

The filter FilterRenkonen

This step removes replicates that are very different to other replicates of the same sample and only makes sense if at least two replicates are used. The Renkonen distance takes into account the whole composition of the replicates.

Renkonen distances (*Renkonen, 1938*) are calculated between each pair of replicates within each sample. Then a cutoff distance is set at a quantile of all renkonen distances of the dataset. This quantile can be given by the user through the **renkonen_distance_quantile** parameter and takes 0.9 (Ninth decile) as default value. Replicates above the quantile distance are removed.

FilterIndel

This filter makes only sense for coding sequences and can be skipped with the parameter **skip_filter_indel**.

It is based on the idea that sequences with indels of 3 nucleotides or its multiples are viable, but all others have frame-shift mutations and are unlikely to come from correct, functional sequences. Therefore, the *modulo* 3 of the length of each variant is determined. The majority of the variants length will have the same modulo 3. All other variants are discarded.

FilterCodonStop

This filter makes only sense for coding sequences and can be skipped with the parameter **skip_filter_codon_stop**.

Given the appropriate genetic code, the presence of codon stops is checked in all reading frames of the direct strand. Variants that have a codon stop in all reading frames are discarded.

When the Genetic code cannot be determined in advance (community includes species from groups using different genetic codes) we suggest using the Invertebrate Mithochondrial Code **genetic_table_number 5**, since its codons STOPs (TAA, TAG) are also codon STOP in almost all genetic codes.

MakeAsvTable

The ASVs that passed all filters (Table *FilterCodonStop* in the SQLITE database) are written to an ASV table with variants in rows, samples in columns and read count per variants per samples are in cells. The read count per sample is the sum of read counts of the replicates. Replicates are not shown.

Additional columns include marker and run names, sequence length, total read counts for each variant, flags on *chimera borderline* information on clustering and expected variants in mock samples (optional).

Clustering information:

When creating an ASV table, all variants are clustered using a user defined identity cutoff (**cluster_identity**; 0.97 by default). The **clusterid** is the centroid, and the number of variants are given in the **clustersize** column. This clustering is a simple help for the users to identify similar variants, but it is NOT part of the filtering process. Clusters of different ASV tables are not directly comparable.

Expected variants in mock samples:

If mock samples have many expected variants or there are several different mock samples, it can be a bit difficult to identify all 'keep' and 'delete' *occurrences*. The **known_occurrences** option in filter command can help you in this task.

First, identify 'keep' occurrences in the mock samples, and make a *known_keep_occurences.tsv* file. Then run the filter command with the **known_occurrences** option. This will add an extra column foreach mock sample in the ASV table. These columns will contain 1 if the variant is expected in the mock sample. From this updated ASV table it will be easier to select 'delete' occurrences with the help of a tableur (LibreOffice, Excel).

```
vtam filter --db db.sqlite --sortedinfo sortedinfo.tsv --sorteddir sorted --asvtable␣
↪asvtable_default.tsv -v --log vtam.log --known_occurrences known_keep_occurrences.
↪tsv
```

The example of of the completed ASV table looks like this:

```
run    marker    variant    sequence_length    read_count    tpos1_run1    tnegtag_
↪run1    14ben01    14ben02    keep_tpos1_run1    clusterid    clustersize    ␣
↪chimera_borderline    sequence
run1    MFZR    25    181    478    478    0    0    0    0    25    1    False    ␣
↪ACTATACCTTATCTTCGCAGTATTCTCAGGAATGCTAGGAACTGCTTTTAGTGTTCTTATTCGAATGGAACTAACATCTCCAGGTGTACAATACCTAC
run1    MFZR    51    181    165    0    0    0    165    0    51    1    False    ␣
↪ACTATATTTAATTTTTGCTGCAATTTCTGGTGTAGCAGGAACTACGCTTTCATTGTTTATTAGAGCTACATTAGCGACACCAAATTCTGGTGTTTTAG
run1    MFZR    88    175    640    640    0    0    0    1    88    1    False    ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCG
```

This step is particularly useful when analysing several runs with the same mock samples, since in the case the same *known_keep_occurrences.tsv* can be used for all runs.

1.4.6 The command taxassign

All variants retained at the end of the filtering steps undergo a taxonomic assignation with this command.

A quick introduction to the **vtam taxassign** command is given in the tutorial *taxassign: Assign variants of ASV table to taxa*.

The command line arguments of the **taxassign** command can be obtained with

```
vtam taxassign --help
```

The most important arguments are these inputs and outputs:

Inputs:

- *asvtable*: TSV file containing variants in the last column (using 'sequence' as a header). Typically it is an ASV table.
- **sorteddir**: Directory containing the demultiplexed fasta files.

Database:

- *db*: Name of an sqlite db. It is used to store and access information on runs, markers, samples, variants, filtering steps and taxonomic assignations. It is created by the **filter** command if it does not exist and it is completed if it exists already.
- *taxonomy*: TSV file containing taxonomic information <LINK to the taxonomy.tsv>
- *blastdbdir*: directory containing the BLAST database <LINK to BLAST batabase>
- *blastdbname*: name of the BLAST database <LINK to BLAST batabase>

Outputs:

- *output*: The input TSV file completed by taxonomic assignations

Variants are BLASTed against the NCBI nt or custom BLAST database. Taxa name and rank is chosen based on lineages of the best hits. Lineages are constructed based on a taxonomy TSV file (See *Reference section*).

For a given %identity between the variant and the *hits*, select hits with *coverage* >= **min_query_coverage**. Depending on the %identity and the **ltg_rule_threshold**, there are two possible rules:

1. %identity>= **ltg_rule_threshold**: Determine the Lowest Taxonomic Group (LTG)<link to the glossary>: Take the Lowest Taxonomic Group that contains at least **include_prop** % of the hits. Otherwise the LTG is not inferred at that %identity level.
2. %identity< **ltg_rule_threshold**: Determine the Lowest Taxonomic Group (LTG)<link to the glossary> (using the same rule as previously) only if selected hits contain at least **min_number_of_taxa** different taxa. Otherwise the LTG is not inferred at that %identity level.

VTAM intends to establish LTG by using first a high %identity (100%). If LTG cannot be defined with this %identity, the %identity is decreased gradually (100%, 99%, 97%, 95%, 90%, 85%, 80%, 75%, 70%) till an LTG can be established. For each variant, the %identity used to infer the LTG is also given in the output. These values should not be ignored. If LTG could only be defined at 80% or lower identity level, the results are not very robust.

Let's see two examples using default values:

Example 1 of taxonomic assignation

A variant has produced 47 hits to *Beatis rhodani* sequences and 1 to a *Baetis cf. rhodani* sequence at 100 %identity. The ltg_rule_threshold is 97%.

Taxon	Nb of hits	%identity	%coverage	TaxID
Baetis rhodani	47	100	100	189839
Baetis cf. rhodani	1	100	100	1469487

For all these alignments, the coverage was above 80 (**min_query_coverage**), so all of them are included in the assignment process. LTG can be determined at the 100% identity level. Since 47 out of 48 sequences are from Baetis rhodani (47/48 > 90 (**include_prop**)), the LTG is *Baetis rhodani*.

Example 2 of taxonomic assignation

A variant has produced 11 hits to 4 taxa at 91-97 %identity. The ltg_rule_threshold is 97%.

Taxon	Nb of hits	%identity	%coverage	TaxID	Lineage
Procloeon pulchrum	2	96	90	1592912	…Ephemeroptera; Pisciforma; Baetidae; Procloeon
Procloeon rivulare	6	90-93	100	603535	…Ephemeroptera; Pisciforma; Baetidae; Procloeon
Baetidae sp. BOLD:ACL6167	1	91	100	1808092	… Ephemeroptera; Pisciforma; Baetidae; unclassified Baetidae
Procloeon sp. BOLD:AAK9569	2	91	100	1712756	…Ephemeroptera; Pisciforma; Baetidae; Procloeon

For this variant there are no hits with at least 100% or 97% identity.

There are only 2 hits with at least 95% identity, both coming from the same taxon. Since 95% is below the **ltg_rule_threshold** (97%), at this identity level we need at least 3 different taxa (**min_number_of_taxa**) to derive an LTG. Therefore, VTAM will not infer LTG at the 95% identity level. The idea of not inferring LTG if the % of identity and the number of taxa is low is to avoid assignment of a variant to taxonomic level which is too precise, and probably erroneous.

At 90% identity level, there are 11 hits, coming from 4 taxa (>min_number_of_taxa). All hits have at least 80% query coverage (**min_query_coverage**). All conditions are met to infer LTG, which is Procloeon.

The *Baetidae sp. BOLD:ACL6167* is not included in the LTG, since the other sequences that are all part of *Procloeon* genus represent 10/11>90% (**include_prop**) of the hits. The idea of using **include_prop** as a parameter is to try to avoid partially or erroneously annotated sequences if there is sufficient data to support an assignment with a higher resolution. In this example **Baetidae sp. BOLD:ACL6167** can in fact be a **Procloeon** species, but we do not have this information.

These parameters can be passed in a YML file via the **params** argument (See *Numerical parameter file*).

1.4.7 The command make_known_occurrences (OPTIONNAL)

VTAM can create TSV file containing known and missing occurrences based on an ASV table, a TSV file containing the sample types and a TSV file containing a mock composition.

The know_occurrences file will be used as an input for the optimize command. It lists all delete occurrences and all keep occurreces. The optimize step will use this file to find a parameter combination, that retains all keep occurrences, and eliminates most delete occurrences.

The missing occurrences file is just a feedback for the users. It lists all expected occurrences that are missing from the input asv table.

A quick introduction to the **vtam make_known_occurrences** command is given in the tutorial *make_known_occurrences: Create file containing the known_occurences.tsv to be used as an inut for optimize*.

The command line arguments of the **make_known_occurrences** command can be obtained with

```
vtam make_known_occurrences --help
```

The most important arguments are these inputs and outputs:

Inputs:

- *asvtable*: TSV file containing variants in the last column (using 'sequence' as a header). Typically it is an ASV table.
- *sample_types*: Path to the TSV file containing the sample types
- *mock_composition*: Path to the TSV file containing the mock composition

Outputs:

- known_occurrences: Path to the file containing the known occurrences.
- missing_occurrences: Path to the file containing the missing occurrences

1.4.8 The command optimize

The optimization step aims to find the optimal values for the parameters of the filtering steps. It is based on occurrences that are clearly erroneous (false positives, flagged as 'delete') or clearly expected (flagged as 'keep'). These occurrences are determined by the user. The optimization step will suggest a parameter setting that keeps all occurrences in the dataset flagged as 'keep' and delete most occurrences flagged as 'delete'. A list of known 'keep' and 'delete' occurrences is given in the *kown_occurrences.tsv* input file. The preparation of this file can be facilitated by running the filter command with the **known_occurrences** option (*See Make the ASV table section*).

The 'keep' occurrences are the expected variants in the mock samples.

Delete occurrences are the following :

- All occurences in negative controls.
- Unexpected occurrences in mock samples.
- Some occurrences of variants in real samples can also be flagged 'delete' if there are samples from markedly different habitats (e.g. marine vs. freshwater; the presence of a variant assigned to a freshwater taxon in a marine sample is clearly an error, and the occurrence can be flagged as 'delete').

Running first the **filter** command with default parameters produces an ASV table, where most of the sequence artefacts are eliminated, and therefore the ASV table has a reasonable size to deal with in a spreadsheet. Users should identify 'keep' and 'delete' occurrences from this output before the optimization step.

All optimization steps are run on the original, non-filtered read counts.

A quick introduction to the **optimize** command is given in the tutorial *optimize: Compute optimal filter parameters based on mock and negative samples*.

The command line arguments of the **optimize** command can be obtained with

```
vtam optimize --help
```

These arguments are the most important inputs and outputs:

Inputs:

- *sortedinfo*: TSV with the information about each sorted read file (output of “sortreads“). The content of the file will define the dataset to be used by “optimize“.
- **sorteddir**: Directory containing the demultiplexed fasta files.
- *known_occurrences*: User created TSV file with occurrences clearly identified as ‘delete’, ‘keep or ‘tolerate’ (See manual).

Database:

- *db*: Name of an sqlite db. It is used to store and access information on runs, markers, samples, variants, filtering steps and taxonomic assignations.

Outputs:

- **outdir**: Path to a directory that will contain the following files:

The following files will be written in the “–outdir” directory <link>

- *optimize_lfn_sample_replicate.tsv*
- *optimize_lfn_read_count_and_lfn_variant.tsv OR optimize_lfn_read_count_and_lfn_variant_replicate.tsv*
- *optimize_lfn_variant_specific.tsv OR optimize_lfn_variant_replicate_specific.tsv*
- *optimize_pcr_error.tsv*

The step OptimizePCRError

Optimizing the *pcr_error_var_prop* parameter is based only on the mock sample composition. The idea is to detect unexpected variants highly similar to expected ones within the same sample. The unexpected variant is likely to come from a PCR error and the proportion of their read counts help to choose the value of **pcr_error_var_prop**.

For each mock sample, all occurrences of unexpected variants (all but the ‘keep’) with a single difference to an expected variant (‘keep’) are detected. For each of these unexpected occurrences *N(i_unexpected)j/N(i expected)j* is calculated. The value for **pcr_error_var_prop** should be higher than the maximum of these proportions. The results are sorted by run, marker, and then by *N_ij_unexpected_to_expected_ratio* in decreasing order. Therefore, for each run marker combination, the **pcr_error_var_prop** parameter should be set above the first value.

Example of *optimize_pcr_error.tsv* :

```
run    marker    sample    variant_expected    N_ij_expected    variant_unexpected  ␣
↪N_ij_unexpected    N_ij_unexpected_to_expected_ratio    sequence_expected   ␣
↪sequence_unexpected
run1    MFZR    tpos1_run1    264    3471    1051    63    0.01815039   ␣
↪ACTTTATTTTATTTTTGGTGCTTGATCAGGAATAGTAGGAACTTCTTTAAGAATTCTAATTCGAGCTGAATTAGGTCATGCCGGTTCATTAATTGGAGA
↪  ␣
↪CCTTTATTTTATTTTTGGTGCTTGATCAGGAATAGTAGGAACTTCTTTAAGAATTCTAATTCGAGCTGAATTAGGTCATGCCGGTTCATTAATTGGAGA
run1    MFZR    tpos1_run1    88    640    89    8    0.01250000   ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
↪  ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
```

The step OptimizeLFNsampleReplicate

Optimizing the *lfn_sample_replicate_cutoff* parameter is based only on the mock sample composition.

For each 'keep' variant in all mock samples (*j*), in all replicates (*k*), *N_ijk/N_jk* is calculated and these proportions are ordered increasingly. (*N_jk* is the total number of reads in the sample *j* replikate *k*. It includes all variants, not just the ones marked as keep) The optimal value for **lfn_sample_replicate_cutoff**, should be under the smallest proportion. This ensures keeping of all expected variants by the **LFN_sample_replicate** filter.

In a fairly complex mock sample this smallest proportion rarely exceeds 0.01, but it is more often in the order of 0.001. Avoid setting this value higher than 0.01, since it can eliminate too many real occurrences.

One scenario (quite rare in our experience) is that all keep variants amplify well, and the suggested value for **lfn_sample_replicate_cutoff** is relatively high (>0.01). If the real samples are expected to have a high number of taxa (e.g. > 20), it is better to lower this value (default in 0.001).

Another case is that one or several expected variants have very few reads compared to the total number of reads in a sample-replicate (*N_ijk/N_jk*), while for the majority of the replicates of the same sample *N_ijk/N_jk* is not too low. In the example below, variant 75 in sample Tpos2_prerun and replicate 2 has only 0.01% of the reads, while in the other 2 replicates it is higher than 0.4%. In this case it is better to choose 0.004 as a **lfn_sample_replicate_cutoff**, knowing that the variant is not filtered out in 2 out of the 3 replicates, therefore, it will remain in the sample after pooling the results over replicates. (See *FilterMinReplicateNumber*)

Example of *optimize_lfn_sample_replicate.tsv* :

```
run     marker     sample     replicate     variant_id     N_ijk     N_jk     lfn_sample_
↪replicate: N_ijk/N_jk     round_down     sequence
prerun     MFZR     Tpos2_prerun     2     75     1     10234     0.00010234     0.000100000 ␣
↪ ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
prerun     MFZR     Tpos1_prerun     3     75     42     10414     0.00403303     0.00400000 ␣
↪ ␣
↪ACTATATTTTATTTTTGGGGCTTGATCCGGAATGCTGGGCACCTCTCTAAGCCTTCTAATTCGTGCCGAGCTGGGGCACCCGGGTTCTTTAATTGGCGA
prerun     MFZR     Tpos2_prerun     3     75     63     15038     0.00418939     0.00410000
```

The steps OptimizeLFNReadCountAndLFNvariant and OptimizeLFNReadCountAndLFNvariantReplicate

These steps find the combinations **lfn_variant_cutoff/read_count_cutoff** and **lfn_variant_replicate_cutoff/read_count_cutoff** that minimize the number of 'delete' occurrences while keeping all 'keep' occurrences.

The **fitler_lfn_variant** and **filter_lfn_variant_replicate** are alternatives around the same idea: Filtering occurrences in function of the their read count in the sample-replicate compared to the total number of reads of the variant in the run (*N_ijk/N_i*; **filter_lfn_variant**) or in the replicate (*N_ijk/N_ik*; **filter_lfn_variant_replicate**). The command **optimize** can have **lfn_variant** or **lfn_variant replicate** mode. Just like for the **filter** command, the default is the **lfn_variant** mode and the **lfn_variant_replicate** mode can be activated by the **lfn_variant_replicate** flag in the command line (see <link>). For simplicity, we will use **filter_lfn_variant** in the rest of this section.

All **FilterLFN** steps and **FilterMinReplicateNumber** are run on the original non-filtered data using a large number of combinations of **lfn_variant_cutoff** and **read_count_cutoff** (all other parameters are default). The values for these two thresholds vary between their default value till the highest value that keeps all 'keep' occurrences. For each combination, the number of 'delete' occurrences remaining in the dataset are counted (nb_delete) and printed to a spreadsheet in increasing order. Users should choose the parameter combination with lowest nb_delete.

Example of *optimize_lfn_read_count_and_lfn_variant.tsv*:

```
occurrence_nb_keep    occurrence_nb_delete    lfn_nijk_cutoff    lfn_variant_cutoff  ␣
↪ run    marker
6    4    73    0.001    run1    MFZR
6    4    73    0.005    run1    MFZR
```

1.4.9 The command pool

This command will pool the results of several run-marker combinations into one ASV table. Variants identical on their identical regions are regrouped to the same line.

Inputs:

- *run_marker*: TSV file listing all run marker combinations to be pooled

Database:

- *db*: Name of an sqlite db. It is used to store and access information on runs, markers, samples, variants, filtering steps and taxonomic assignations. It is created by **filter** if it does not exist and it is completed if it exists already.

Outputs:

- *asvtable*: Name of the pooled ASV table

In order to increase the chance of amplifying a large number of taxa, more than one primer pair (markers) can be used to amplify the same locus. The annealing sites of different markers can be slightly different, making the direct pooling of the results of different markers impossible. This step is carried out using the **vsearch –cluster_size** command with 1 as identity cutoff. In this way only variants identical in their overlapping regions are pooled together.

As for making ASV tables after filtering, all variants are also clustered using a user defined identity cutoff (**cluster_identity**; 097 by default). The **clusterid** is the centroid, and the number of variants are given in the **clustersize** column. This clustering is a simple help for the users to identify similar variants, but it is NOT part of the filtering process nor part of pooling identical variants on their overlapping regions. Clusters of different ASV tables are not directly comparable.

The output table contains the following columns:

- variant_id: ID of a variant representative of the variants identical on their overlapping region
- pooled_variants: List of variant identical on their overlapping region
- run
- marker
- [one column per sample] : presence(1)/absence(0)
- clusterid: Certoïd of a cluster (cluster_identity threshold)
- clustersize: Number of variants in the cluster
- pooled_sequences: Comma separated list of variants identical on their overlapping region
- sequence: Sequence of the representative variant

1.4.10 The command taxonomy and the taxonomic lineage input

VTAM requires a taxonomical file in TSV format that looks like this:

```
tax_id    parent_tax_id    rank    name_txt    old_tax_id
1    1    no rank    root
2    131567    superkingdom    Bacteria
6    335928    genus    Azorhizobium
7    6    species    Azorhizobium caulinodans    395
9    32199    species    Buchnera aphidicola    28241
10    1706371    genus    Cellvibrio
```

These are the columns of the file:

- tax_id: Integer with the NCBI taxonomic ID.
- parent_tax_id: Integer with the parent's NCBI taxonomic ID.
- rank: String with the taxonomic rank.
- name_txt: String with the scientific name.
- old_tax_id: Optional. Integer with a previous taxonomic ID that will be tried if an entry was not found in the tax_id column.

A precomputed file can be downloaded using this command. This version in not necessarily the most up to date compared to the ncbi taxonomy database, but it works with our custom database:

```
vtam taxonomy --output taxonomy.tsv --precomputed
```

Alternatively, the command will download the up-to-date ncbi taxonomy database (https://www.ncbi.nlm.nih.gov/taxonomy) and create a fresh TSV file with the latest data in NCBI. This is strongly recommended if you are using a recently downloaded version of the ncbi_nt:

```
vtam taxonomy --output taxonomy.tsv
```

This step can take several minutes. Make sure you have a steady internet connection.

The *taxonomy.tsv* file created by the script is ready to use as is if you intend to use the full NCBI nucleotide *BLAST database* or our *precomputed non-redundant database specific to COI*. However, if you create a custom database, containing sequences from taxa not yet included in NCBI taxonomic database, you have to complete this file with arbitrary Taxonomic IDs used in your custom database and link them to existing NCBI taxids. We suggest using negative TaxIDs for taxa not present in NCBI Taxonomy database.

An example is found here:

```
tax_id    parent_tax_id    rank    name_txt    old_tax_id
...
233297    34643    genus    Chrysso    449633
41073    535382    family    Carabidae
...
-1    233297    species    Chrysso pelyx
-2    41073    genus    Pelophila
-3    -2    species    Pelophila borealis
```

1.4.11 The BLAST database

VTAM uses a BLAST database for taxonomic assignment. There are three possibilities:

- Download the full NCBI nucleotide (NCBI_nt) BLAST (ca. 100 Gb).
- Download a precomputed non-redundant database specific to the first half (600-700bp) of the COI gene (less than 1 Gb).

- Build a custom BLAST database with local sequences.

NCBI nt BLAST database

The NCBI nucleotide (Genbank) database can be downloaded with the following commands:

```
wget "ftp://ftp.ncbi.nlm.nih.gov/blast/db/nt*"
md5sum --check nt.*.tar.gz.md5
tar -zxvf nt.*.tar.gz
rm nt.*.tar.gz*
```

It contains all NCBI nucleotide sequences and thus, it is not limited to COI.

Precomputed non-redundant COI database

We have created a non-redundant database specific to the first half (600-700bp) of the COI gene. It contains COI sequences from NCBI nt and BOLD, they cover at least 80% of the barcoding region of the COI. Identical sequences of the same taxon are present only once in the database. The latest version of this database can be downloaded with the following command:

```
vtam coi_blast_db --blastdbdir vtam_db/coi_blast_db
```

Earlier versions can be downloaded by specifying the name of database:

```
vtam coi_blast_db --blastdbdir vtam_db/coi_blast_db --blastdbname coi_blast_db_
↪20200420
```

The available versions are found here: https://github.com/aitgon/vtam/releases/latest

Custom database

To create a custom database, make sure that you have installed blast>=2.9.0 available in Conda.

Then create a fasta file *mydb.fas* and a mapping *taxid_map.tsv* from sequences to NCBI taxa. These are the first lines of the *mydb.fas* file.

```
>MG1234511
AACACTTTATTTTATTTTTGGAATTTGAGCTGGAATAGTAGGAACATCATTAAGAATTTTAATTCGATTGGAATTAAGAACAATTTCTAATTTAATTGGA
>MG1588471_1
AACATTATATTTTATTTTTGGTGCTTGATCAAGAATAGTGGGGACTTCTTTAAGAATACTTATTCGAGCTGAATTAGGGTGTCCGAATGCTTTAATTGGG
>MG1588471_2
AACATTATATTTTATTTTTGGTGCTTGATCAAGAATAGTGGGGACTTCTTTAAGAATACTTATTCGAGCTGAATTAGGGTGTCCGAATGCTTTAATTGGG
>LOCAL_SEQ1
TATTTTATTTTTGGAATATGAGCAGGAATATTAGGATCATCAATAAGATTAATTATTCGAATAGAACTAGGTAACCCTGGATTTTTAATTAATAATGATC
```

These are the first lines of the *taxid_map.tsv* file where the first column is the sequence Identifier and the second is the NCBI TaxID of the taxon of origin or arbitrary unique TaxIDs included in the *taxonomy.tsv* file:

```
MG1234511    2384799
MG1588471_1    2416844
MG1588471_2    2416875
LOCAL_SEQ1    -3
```

Then you run this command:

```
makeblastdb -in mydb.fas -parse_seqids -dbtype nucl -out nt -taxid_map taxid_map.tsv
```

1.4.12 Traceability

Variants, samples, the results of filtering steps, and the taxonomic assignments are stored in a sqlite database. This provides a possibility to trace the history of the analyses. You can easily discover the sqlite database with a sqlite browser program (For example https://sqlitebrowser.org/ or https://sqlitestudio.pl).

Here we propose a few examples of sql commands that can be adapted to extract the information you need.

Each filter has a table in the database, and they contain the following fields:

- run_id
- marker_id
- varinat_id
- sample_id
- replicate
- read_count
- filter_delete

The FilterLNF table is special, because it is composed of several filters: filter_id=2, 3, … To select variants that passed all filters, we need to check filter_id=8:

Count the number of variants after FilterChimera for run 1 and marker 1

```
select count(distinct variant_id) from FilterChimera where run_id=1 and marker_id=1␣
↪and filter_delete=0
# for FilterLFN
select count(distinct variant_id) from FilterLFN where run_id=1 and marker_id=1 and␣
↪filter_delete=0 and filter_id=8
```

Count the number of non_empty samples after FilterChimera for run 1

```
select count(distinct sample_id) from FilterChimera where run_id=1 and filter_delete=0
# for FilterLFN
select count(distinct sample_id) from FilterLFN where run_id=1 and filter_delete=0␣
↪and filter_id=8
```

Count the number of reads that passed the filter

```
select sum(read_count) from FilterChimera where run_id=1 and filter_delete=0
# for FilterLFN
select sum(read_count) from FilterLFN where run_id=1 and filter_delete=0 and filter_
↪id=8
```

Get the list of the samples after a given filtering step

```
select distinct Sample.name as sample from FilterChimera, Sample where FilterChimera.
↪sample_id=Sample.id and filter_delete=0 order by Sample.name
# for FilterLFN
select distinct Sample.name as sample from FilterLFN, Sample where FilterLFN.sample_
↪id=Sample.id and filter_delete=0 and filter_id=8 order by Sample.name
```

Get the list of the sample-replicates after a given filtering step

```
select distinct Sample.name as sample, FilterChimera.replicate from FilterChimera,␣
↪Sample where FilterChimera.sample_id=Sample.id and filter_delete=0 order by Sample.
↪name, FilterChimera.replicate
# for FilterLFN
select distinct Sample.name as sample, FilterLFN.replicate from FilterLFN, Sample␣
↪where FilterLFN.sample_id=Sample.id and filter_delete=0 and filter_id=8 order by␣
↪Sample.name, FilterLFN.replicate
```

Get the list of the variants after a given filtering step

```
select distinct Variant.id, Variant.sequence from FilterChimera, Variant where␣
↪FilterChimera.variant_id=Variant.id and filter_delete=0 order by Variant.id
# for FilterLFN
select distinct Variant.id, Variant.sequence from FilterLFN, Variant where FilterLFN.
↪variant_id=Variant.id and filter_delete=0 and filter_id=8 order by Variant.id
```

1.5 Input/Output Files

Help on the usage and complete list of I/O arguments of each command can be obtained using the command line help

```
vtam COMMAND --help
i.e.
vtam filter --help
```

Here we detail the content of the I/O files

1.5.1 params

Input of most commands. YML file with *numerical parameters*. Can be omitted if all parameters are by default. Simple text file with a "parameter name: parameter value" format. One parameter per line e.g.

```
lfn_variant_cutoff: 0.001
lfn_sample_replicate_cutoff: 0.003
lfn_read_count_cutoff: 70
pcr_error_var_prop: 0.05
```

1.5.2 fastqinfo

Input of *merge*. TSV file with the following columns:

- TagFwd: Sequence of the tag on the forward primer (5'=>3')
- PrimerFwd: Sequence of the forward primer (5'=>3')
- TagRev: Sequence of the tag on the reverse primer (5'=>3')
- PrimerRev: Sequence of the reverse primer (5'=>3')
- Marker: Name of the marker (e.g. MFZR)
- Sample: Name of the sample
- Replicate: ID of the replicate
- Run: Name of the sequencing run

- FastqFwd: Name of the forward fastq file
- FastqRev: Name of the reverse fastq file

1.5.3 fastainfo

Output of *merge*, input of *sortreads*. TSV file with the following columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- sample: Name of the sample
- replicate: ID of the replicate
- tagfwd: Sequence of the tag on the forward primer (5'=>3')
- primerfwd: Sequence of the forward primer (5'=>3')
- tagrev: Sequence of the tag on the revrese primer (5'=>3')
- primerrev: Sequence of the reverse primer (5'=>3')
- mergedfasta: name of the fasta file with merged sequences

1.5.4 sortedinfo

Output of *sortreads*, input of *filter* and optimize. TSV file with the following columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- sample: Name of the sample
- replicate: ID of the replicate
- sortedfasta: name of the fasta file containing merged, demultiplexed, trimmed sequeces

1.5.5 db

I/O of *filter*, *taxassign*. Input of optimize, *pool*. Sqlite database containing variants, samples, replicates, read counts, information on filtering steps, taxonomic assignations.

1.5.6 asvtable

Output of *filter* or *pool*, input of *taxassign*. TSV file with the variants (in lines) that passed all filtering steps, samples (in columns), presence-absence (output of pool) or read counts (output of filter) in cells and additional columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- variant: Variant ID
- pooled_variants (only in output of pool): IDs of variants pooled since identical in their overlapping regions
- sequence_length length of the variant
- read_count: Total number of reads of the variants in the samples listed in the table

- [one column per sample] : presence-absence (output of pool) or read counts (output of filter)
- clusterid: ID of the centroïd of the cluster (0.97 clustering of all variants of the asv table)
- clustersize: Number of variants in the cluster
- chimera_borderline (only in output of filter): Potential chimeras (very similar to one of the parental sequence)
- [keep_mockXX; One column per mock sample, if known_occurrences option is used]: 1 if variant is expected in the mock sample, 0 otherwise
- pooled_sequences (only in output of pool): Sequences of pooled_variants
- sequence: Sequence of the variant

1.5.7 known_occurrences

Input of *filter* and optimize. Output of make_known_occurrences. TSV file with expected occurrences (keep) and known false positives (delete).

- Marker: Name of the marker (e.g. MFZR)
- Run: Name of the sequencing run
- Sample: Name of the sample
- Mock: 1 if sample is a mock, 0 otherwise
- Variant: Varinat ID (can be empty)
- Action: keep (occurrences that should be kept after filtering) or delete (clear false positives)
- Sequence: Sequence of the variant
- Tax_name: optional, not used by optimize

1.5.8 mock_composition

Input of filter. TSV file with expected sequences in mock samples.

- Marker: Name of the marker (e.g. MFZR)
- Run: Name of the sequencing run
- Sample: Name of the sample
- Mock: 1 if sample is a mock, 0 otherwise
- Variant: Variant ID (can be empty)
- Action: keep (occurrences that should be kept after filtering) or delete (clear false positives) or tolerate (variant present in a mock sample but amplifies badly)
- Sequence: Sequence of the variant
- Tax_name: optional, not used by optimize

1.5.9 sample_types

Input of make_known_occurrences. TSV file.

- run: Name of the sequencing run

- **sample: Name of the sample**
  - sample_type: real/negative(negative control)/mock
  - habitat: habitat type (e.g. freshwater, marine), NA for negative contol samples. It is used to detect occurrences that do not correspond to the habitat type.

1.5.10 missing_occurrences

Output of make_known_occurrences. TSV file with keep occurrences that are missing from the input ASV table.

- Marker: Name of the marker (e.g. MFZR)
- Run: Name of the sequencing run
- Sample: Name of the sample
- Mock: 1 if sample is a mock, 0 otherwise
- Variant: Variant ID (can be empty)
- Action: keep (occurrences that should be kept after filtering) or delete (clear false positives)
- Sequence: Sequence of the variant
- Tax_name: optional, not used by optimize

1.5.11 optimize_lfn_sample_replicate.tsv

Output of optimize. TSV file with the following columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- sample: Name of the sample
- replicate: ID of the replicate
- variant: Variant ID
- N_ijk: Number of reads of variant i, in sample j and replicate k
- N_jk: Number of reads in sample j and replicate k (all variants)
- N_ijk/N_jk
- round_down: Rounded value of N_ijk/N_jk
- sequence: Variant sequence

1.5.12 optimize_lfn_read_count_and_lfn_variant.tsv OR optimize_lfn_read_count_and_lfn_variant_replicate.tsv

Output of optimize. TSV file with the following columns:

- occurrence_nb_keep: Number of keep occurrence left after filtering with lfn_nijk_cutoff and lfn_variant_cutoff values
- occurrence_nb_delete: Number of delete occurrence left after filtering with lfn_nijk_cutoff and lfn_variant_cutoff values
- lfn_nijk_cutoff: lfn_read_count_cutoff

- lfn_variant_cutoff or lfn_variant_replicate_cutoff
- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)

1.5.13 optimize_lfn_variant_specific.tsv OR optimize_lfn_variant_replicate_specific.tsv

Output of optimize. TSV file with the following columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- variant: Variant ID
- replicate: (if optimize_lfn_variant_replicate_specific.tsv) ID of the replicate
- action: Type d'occurrece (delete/keep)
- read_count_max: Max of N_ijk for a given i
- N_i (optimize_lfn_variant_specific.tsv) : Number of reads of variant i
- N_ik (optimize_lfn_variant_replicate_specific.tsv): Number of reads of variant i in replicate k
- lfn_variant_cutoff: read_count_max/N_i or read_count_max/N_ik
- sequence: Variant sequence

1.5.14 optimize_pcr_error.tsv

Output of optimize. TSV file with the following columns:

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)
- sample: Name of the sample
- variant_expected: ID of a keep variant
- N_ij_expected: Number of reads of the expected variant in the sample (all replicates)
- variant_unexpected: ID of an unexpected variants with one mismatch to the keep variant
- N_ij_unexpected: Number of reads of the unexpected variant in the sample (all replicates)
- N_ij_unexpected_to_expected_ratio: N_ij_unexpected/N_ij_expected
- sequence_expected: Sequence of the expected variant
- sequence_unexpected: Sequence of the unexpected variant

1.5.15 output (taxassign)

Output of *taxassign* The input asvtable completed with the following columns:

- ltg_tax_id: TaxID of the LTG (Lowest Taxonomic Group)
- ltg_tax_name ltg_rank: Name of the LTG
- identity: Percentage of identity used to determine the LTG

- blast_db: Name of the taxonomic BLAST database files (without extensions)
- phylum: Phylum of LTG
- class: class of LTG
- order: order of LTG
- family: family of LTG
- genus: genus of LTG
- species: species of LTG

1.5.16 taxonomy

Output of *taxonomy*, input of *taxassign*. TSV file with information of all taxa in the reference (BLAST) database.

- tax_id: Taxonomic identifier of the taxon
- parent_tax_id: Taxonomic identifier of the direct parent of the taxon
- rank: Taxonomic rank of the taxon (e.g. class, species, no rank)
- name_txt: Name of the taxon
- old_tax_id: TaxID of taxa merged to taxon (not valid any more)

1.5.17 runmarker

Input of *pool*. TSV file with the list of all run-marker combinations to be pooled.

- run: Name of the sequencing run
- marker: Name of the marker (e.g. MFZR)

1.6 Glossary

Terms are defined as they are used in VTAM. They might not hold or be sufficiently precise in another context outside VTAM.

1.6.1 ASV or variant

Amplicon Sequence Variant: Unique amplicon sequence (Callahan et al., 2017). Identical reads are pooled into a variant. Variants are characterized by the number of reads in each replicate of each sample, which we also call “sample-replicate”.

1.6.2 ASV table

Representation of presence of each of the variants in each sample; Variants are in lines, samples are in columns, read numbers or presence absence are in cells.

1.6.3 BLAST hit

A sequence from the BLAST database that has significant similarity to the query sequence (variant).

1.6.4 Chimera borderline

When the chimera formation happens near the extremity of the parental sequences, the resulting chimera is very similar to one of the parental sequences. These chimeras are difficult to tell apart from real variants.

1.6.5 Coverage

[in BLAST] the percentage of the length of the query sequence that is covered by the BLAST alignment.

1.6.6 Demultiplexing

Sorting reads to sample-replicates according to the presence of primers and tags at their extremities.

1.6.7 Dereplication

Identical reads are pooled into a variant, and the read count is kept as an information.

1.6.8 Flag

If variants or occurrences are flagged, they remain in the dataset after the corresponding filtering step, but they will be marked (flagged) in the ASV table.

1.6.9 Locus/Gene

genomic region (COI, LSU, MatK, RBCL. . .)

1.6.10 Lowest Taxonomic Group (LTG)

The taxonomic group of the highest resolution (species is high resolution, phylum is low), that contains all or a given % of the sequences.

1.6.11 Marker

A region amplified by one primer pair.

1.6.12 Merge

Assemble each forward and reverse reads (read pair) to a single sequence.

1.6.13 Mock sample

Sample with known DNA composition.

1.6.14 Modulo

Remainder after a division of one number by another. (e.g. the modulo 3 of 4 is 1)

1.6.15 Occurrence

Presence of a variant in a sample or sample-replicate

Unexpected or 'delete' occurrence

A variant in a sample that is known to be erroneous. It can be a variant in a negative control, an unexpected variant in a mock sample, or variant identified from a clearly different habitat than that of the sample.

Expected or 'keep' occurrence

A variant that should be present in the given sample after filtering. These are expected variants in the mock samples.

1.6.16 OLSP

One-Locus-Several-Primers Strategy of using more than one primer pairs that amplify the same locus (slight variation in the position of the annealing sites) in order to increase the taxonomic coverage (Corse et al., 2019).

1.6.17 Renkonen distance

1 - sum(p1i, p2i), where p1i is the frequency of variant i in sample-replicate1 (Renkonen, 1938)

1.6.18 Replicate

or Replicate series : Pool of a single replicate from each sample of a run.

1.6.19 Run

A pool of samples and the associated positive (mock) and negative controls. Ideally, they are obtained in the same sequencing run.

1.6.20 Sample

DNA extraction from a given environment/individual.

1.6.21 Sample-Replicate

Technical replicate of the same sample. e.g different PCR on the same DNA extraction.

1.6.22 Tag-jump

Generation of artefactual sequences in which amplicons carry different tags than originally applied (Schnell et al., 2015)

1.6.23 Tag

Short DNA sequences present at one or both extremities of the amplified DNA fragment. A tag or the combination of forward and reverse tags determine the sample-replicate where the read comes from.

1.6.24 Trimming

Removing part of the extremities of a sequence (e.g. trim the tags/adapters/primers from a read to obtain the biological sequence)

1.6.25 TSV

A text file format with tab-separated values.

1.7 List of References

Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J. H., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research, 25(17), 3389–3402. https://doi.org/10.1093/nar/25.17.3389

Callahan, B.J., McMurdie, P.J., Holmes, S.P., 2017. Exact sequence variants should replace operational taxonomic units in marker-gene data analysis. ISME J. 11, 2639–2643. https://doi.org/10.1038/ismej.2017.119

Corse, E., Tougard, C., Archambaud-Suard, G., Agnèse, J.-F., Mandeng, F.D.M., Bilong, C.F.B., Duneau, D., Zinger, L., Chappaz, R., Xu, C.C.Y., Meglécz, E., Dubut, V., 2019. One-locus-several-primers: A strategy to improve the taxonomic and haplotypic coverage in diet metabarcoding studies. Ecol. Evol. 9, 4603–4620. https://doi.org/10.1002/ece3.5063

Martin, M. (2011). Cutadapt removes adapter sequences from high-throughput sequencing reads. EMBnet.Journal, 17(1), 10–12. https://doi.org/10.14806/ej.17.1.200

Renkonen, O., 1938. Statistisch-Okologische Untersuchungen uber die terrestische Kaferwelt der finnischen Bruchmoore. Zool Soc Zool-Bot Fenn Vanamo 6, 1–231.

Rognes, T., Flouri, T., Nichols, B., Quince, C., Mahé, F., 2016. VSEARCH: a versatile open source tool for metagenomics (No. e2409v1). PeerJ Preprints. https://doi.org/10.7287/peerj.preprints.2409v1

Schnell, I.B., Bohmann, K., Gilbert, M.T.P., 2015. Tag jumps illuminated – reducing sequence-to-sample misidentifications in metabarcoding studies. Mol. Ecol. Resour. 15, 1289–1303. https://doi.org/10.1111/1755-0998.12402

1.8 Citation

González, A., Dubut, V., Corse, E., Mekdad, R., Dechatre, T. and Meglécz, E.. *VTAM: A robust pipeline for processing metabarcoding data using internal controls*. bioRxiv: 10.1101/2020.11.06.371187v1.

1.9 Change Log

Changes In Version 0.2.0 (May 12, 2022)

- **Addressed issues 12, 14, and 19: refactored commands merge sortreads and filter to work with zipped files (.gz and .bz2 fo**

  - Merge and sortreads produce zipped files when zipped files are provided: issues
  - filter can accept compressed input: issue 19,
  - sortreads produce compressed output when provided with compressed inputs: issue 14,
  - merge produce compressed output when provided with compressed: inputs 12.
- **Addressed issues 15, 16, 17, 18 (improved demultiplexing with sortreads):**
  - implemented –no_reverse option to check only on strand: issue 15,
  - use cutadapt v3 to search all tags pairs in parallel: issue 16,
  - added options –tag_to_end and –primer_to_end to search and trim for tags (issue 17) and primers (issue 18) only at the end of the strands.
- **Addressed issue 20:**
  - added command random_seq to produce random samples from a dataset
- **Addressed issue 1:**
  - Integrated the make_known_occurrences command to create a .tsv file with the known occurrences.

Changes In Version 0.1.22 (Jan 31, 2022)

- Updated and tested with python 3.9 and 3.10

Changes In Version 0.1.21 (Dec 11, 2020)

- ENH added '–countreads' option to 'vtam pool'
- ENH using OSF database for VTAM files
- DOC
- BUG bugs

Changes In Version 0.1.20 (Oct 15, 2020)

- DOC
- ENH Keep separated samples from different run in pooled asv table
- BUG

Changes In Version 0.1.19 (Oct 10, 2020)

- BUG fixed filter codon stop and wrapper tests
- ENH Fixed used of LFS resulting in github fees

Changes In Version 0.1.18 (Sep 23, 2020)

- BUG bugs fixed

Changes In Version 0.1.17 (Sep 19, 2020)

- DOC Renamed and improved command-line interface help
- ENH Added a 'vtam example' command to generate a file tree for a quick start

Changes In Version 0.1.16 (Sep 12, 2020)

- DOC updated docs and doc files

Changes In Version 0.1.15 (Sep 10, 2020)

- BUG Compatible biopython >= 1.78

Changes In Version 0.1.14 (Sep 8, 2020)

- ENH Cluster sequences in ASV table
- ENH Label keep occurrences in ASV table
- RFR Renamed biosample to sample

Changes In Version 0.1.13 (Jul 12, 2020)

- BUG fixed read fasta from sorted reads
- ENH partially windows compatible

Changes In Version 0.1.12 (Jul 3, 2020)

- BUG fixed FilterPCRerror runned with all variants

Changes In Version 0.1.11 (Jun 22, 2020)

- ENH verification of –cutoff_specific and –lfn_variant_replicate arguments

Changes In Version 0.1.10 (Jun 18, 2020)

- RFR Refactored optimize lfn variant or variant replicate
- TST New tests optimize lfn variant or variant replicate

Changes In Version 0.1.9 (Jun 12, 2020)

- BLD requirements.txt missing in build

Changes In Version 0.1.8 (Jun 12, 2020)

- BUG Fixed that vtam must run again if –params file is updated (Mantis issues 0002620, 0002621)
- ENH vtam coi_blast_db progressbar

Changes In Version 0.1.7 (Jun 8, 2020)

- RFR vtam coi_blast_db refactor/added –blastdbdir BLASTDBDIR and –blastdbname BLASTDBNAME arguments

Changes In Version 0.1.6 (Jun 3, 2020)

- BUG 0002609 Fixed lfn_variant(_replicate) cutoff specific
- BUG Various bugs

Changes In Version 0.1.5 (Mai 31, 2020)

- ENH: issue 0002607. check paramater names in params.yml
- BUG: issue 0002606. When using –params in filter lfn_read_count_cutoff takes the value of lfn_sample_replicate_cutoff

Changes In Version 0.1.4 (Mai 29, 2020)

- BUG: Fixed filter

Changes In Version 0.1.3 (Mai 27, 2020)

- BUG: Fixed OptimizePCRerror

Changes In Version 0.1.2 (Mai 25, 2020)

- Bug FilterMinReplicateNumber fixed

Changes In Version 0.1.1 (Mai 24, 2020)

- Added tests

- Refactored code
- Change optimization based on explicit ‘keep’ and ‘delete’ variants

Changes In Version 0.1.0 (April 30, 2020)

- Bugs fixed
- Added tests
- Created test dataset
- Made faster some parts of the code

Changes In Version 0.0.1.4 (April 13, 2020)

- Bugs fixed
- Updated to use uchime3_denovo
- Updated to use wopmars 11

Changes In Version 0.0.1.3 (April 7, 2020)

- VariantReadCount: Fixed “insert variants”

Changes In Version 0.0.1.2 (April 5, 2020)

- ‘sortreads’ based on cutadapt
- ‘filter’ commands output to asvtable file instead to output directory
- new ‘global_read_count_threshold’ that will stop variants below this parameter to entering the database

Changes In Version 0.0.1.1 (March 22, 2020)

- Change subcommand “poolmarkers” to “pool”
- Reorder optimize columns and other minor output improvements
- Fixed FilterLFNreference
- Renkonen filter does not run if only one replicate

Changes In Version 0.0.1 (March 18, 2020)

- First version running until the end without apparent bugs affecting results

1.10 Contributor Guide

The docstring format is ‘numpy’. Examples are given here: https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html

CHAPTER 2

Indices and tables

- genindex
- modindex
- search