GeFaST user manual

Version: 2.1.0

19 April 2021

Contents

1 Introduction

1.1 What is GeFaST?

GeFaST (**Ge**neralised **Fa**stidious **S**warming **T**ool) is a de novo clustering tool, primarily designed for processing nucleotide sequences (amplicons) obtained from high-throughput sequencing. GeFaST picks up the two-phased, iterative clustering strategy of Swarm and generalises resp. extends both the clustering and the refinement phase.

GeFaST considers the iterative strategy as an abstract scheme, which can be coupled with different notions of distance. The refinement phase has been generalised by broadening the applicability of the original fastidious approach and introducing further refinement methods. Still, the command-line interface and the output behaviour are close to Swarm.

For more information, see the related publications:

- Müller, R., & Nebel, M. (2018). GeFaST: An improved method for OTU assignment by generalising Swarm's fastidious clustering approach. *BMC Bioinformatics*, 19(1), 321. doi:10.1186/s12859-018-2349-1

1.2 Installation

The source code of GeFaST is available on GitHub: `https://github.com/romueller/gefast`.

1.2.1 Dependencies

GeFaST has been developed for Linux and has the following dependencies:

- C++ (GCC 4.9.2 or higher)
- StatsLib C++ library
- GCE-Math C++ library

1.2.2 Building GeFaST

GeFaST comes with a makefile to simplify the building process. In order to build GeFaST, get the source code from GitHub and compile it as follows:

```
git clone https://github.com/romueller/gefast.git
cd gefast

# install StatsLib / GCE-Math library
git clone -b master --single-branch https://github.com/kthohr/stats libs/stats
git clone https://github.com/kthohr/gcem libs/gcem
mkdir libs/gcem/build
cd libs/gcem/build
cmake ..  -DCMAKE_INSTALL_PREFIX=../../stats
make install
cd ../../..

make
```

Above commands create the `GeFaST` binary in the `build/` subdirectory.

2 Usage

GeFaST is controlled via a simple command-line interface:

```
GeFaST <mode> <input> <config> [option]...
```

The first three arguments are mandatory and fixed in their order:

- `<mode>`: The abbreviation of the mode (see below).
- `<input>`: By default, GeFaST expects a comma-separated list of file paths as its second argument. This behaviour can be changed with the `--list_file` option (see the list of input and output options in Section 3.2).
- `<config>`: The file path of the configuration file containing the (basic) configuration for this execution of GeFaST.

These arguments are followed by an arbitrary list of options described in the next section. When the same option is provided several times, the value of the last occurrence is used. When an option is provided in the configuration file and via command line, the value specified on the command line has priority.

Modes. GeFaST offers several modes, which group related clustering techniques and notions of distance or provide specific functionalities (e.g. dereplication). The chosen mode influences the fundamental behaviour of GeFaST as well as the available parameters. Currently, GeFaST offers the following modes:

Mode (abbreviation)	Synopsis
Levenshtein (lev)	Cluster amplicons based on the number of edit operations in optimal pairwise alignments.
Alignment score (as)	Cluster amplicons based on the score of optimal pairwise alignments.
Alignment-free (af)	Cluster amplicons based on distances, which consider numeric features derived from the sequences.
Quality Levenshtein (qlev)	Cluster amplicons based on the number of edit operations in optimal pairwise alignments considering the quality scores associated with the sequences.
Quality alignment score (qas)	Cluster amplicons based on the score of optimal pairwise alignments considering the quality scores associated with the sequences.
Consistency (cons)	Cluster amplicons using a notion of consistency considering the quality and abundance of amplicons.
Dereplication (derep)	Group amplicons based on exact sequence equality.

More details on the different modes are provided in Section 4.

Examples of how to run GeFaST in the different modes are provided in the `examples/` directory of GeFaST's GitHub page (see Section 1.2).

3 General configuration

3.1 Configuration file.

The configuration file is a (possibly empty) plain-text file containing key-value pairs. Each configuration parameter is written in its own line and has the form `<key>=<value>`. Empty lines and comment lines (starting with #) are allowed.

The configuration file allows the specification of extensive configurations that can be reused easily, by adding or changing options selectively through the command line. Specified options which do not apply to the selected mode are simply ignored.

An exemplary configuration file could look like this:

```
# basic configuration
min_length=20
output_otus=otus.txt
```

Based on this configuration file, GeFaST would discard sequences shorter than 20 nucleotides and create the specified OTU output file.

3.2 Options

The following lists provide information on the general options that are available to all modes. Usually, these can be provided as part of the configuration file or via the command line. As mentioned above, when an option is specified in both ways, the value provided on the command line is used.

In addition, there is the **misc** option, which can only be specified in the configuration file. This option is a (by default empty) $-separated list of key-value pairs. Key and value of each pair are separated by a colon.

Input and output options.

-a, --alphabet *string*
Discard sequences which contain characters other than the ones specified in the alphabet string.
Default: `ACGT`
Configuration keyword: `alphabet`

-i, --output_internal *string*
Output the links underlying the clusters as a tab-separated table in the specified file. Each line contains one link represented through the (1) identifier of the parent amplicon, (2) the identifier of the child amplicon, (3) their distance, (4) the cluster identifier, and (5) the generation number of the child amplicon.
Default: ϵ (empty string, do not create this output)
Configuration keyword: `output_internal`

-o, --output_otus *string*
Output the members of the clusters in the specified file. Each line correponds to one cluster and contains its members represented through identifier and abundance. The members are separated via a single space, while identifier and abundance are separated by the symbol specified via the `--sep_abundance` option.
Default: ϵ (empty string, do not create this output)
Configuration keyword: `output_otus`

-s, --output_statistics *string*
Output statistics on the clusters as a tab-separated table in the specified file. Each line describes one cluster through (1) the number of unique sequences, (2) the mass of the cluster, (3) the identifier of the seed, (4) the abundance of the seed, (5) the number of singletons, (6) the number of iterations before the cluster reached its natural limits, and (7) the number of cumulated differences between the seed and the furthermost amplicon. The statistics (6) and (7) are not affected by the refinement phase.
Default: ϵ (empty string, do not create this output)
Configuration keyword: `output_statistics`

-w, --output_seeds *string*

Output the cluster representatives in the specified file using the FASTA format. The description line of each entry consists of the identifier of the representative and the mass of the cluster (sum of the abundances of all its members), separated by the symbol specified via the `--sep_abundance` option. All sequences are normalised to upper case.
Default: ϵ (empty string, do not create this output)
Configuration keyword: `output_seeds`

-u, --output_uclust *string*

Output the clustering results in the specified file using a UCLUST-like format, describing a tab-separated table with ten columns and three different entry types (S, H, C). Each amplicon is either a seed (cluster centroid, S) or a simple member (hit, H). Cluster records (C) provide additional information on each cluster. The columns are filled as follows:

(1) record type (S, H, or C)

(2) cluster number (zero-based)

(3) sequence length (S, H) or cluster size (C)

(4) percentage of similarity with seed based on an alignment (H) or '*' (S, C)

(5) match orientation '+' (H) or '*' (S, C)

(6) not used, set to '*' (S, C) or 0 (H)

(7) not used, set to '*' (S, C) or 0 (H)

(8) CIGAR representation of alignment (H) or '*' (S, C)

(9) identifier and abundance of member (H) or of seed (S, C)

(10) identifier of seed (H) or '*' (S, C)

Default: ϵ (empty string, do not create this output)
Configuration keyword: `output_uclust`

--sep_abundance *string*

Change the separator symbol between the identifier and the abundance of a sequence to the specified string. The separator symbol can consist of more than one character.
Default: `_` (underscore)
Configuration keyword: `sep_abundance`

--min_length *non-negative integer*

Discard all sequences shorter than the specified length.
Default: `0` (deactivated)
Configuration keyword: `min_length`

--max_length *non-negative integer*

Discard all sequences longer than the specified length.
Default: `0` (deactivated)
Configuration keyword: `max_length`

--min_abundance *non-negative integer*

Discard all sequences with an abundance lower than the specified value.
Default: `0` (deactivated)
Configuration keyword: `min_abundance`

--max_abundance *non-negative integer*

Discard all sequences with an abundance higher than the specified value.
Default: `0` (deactivated)
Configuration keyword: `max_abundance`

--mothur

Output the clusters in a format compatible with mothur. The command-line option has no argument, while the configuration keyword requires the value `1` (activated) or `0` (deactivated).
Configuration keyword: `mothur`

--quality_encoding *string*

Use the specified quality encoding when working with sequences supplemented by quality scores. GeFaST supports the following encodings:

- `sanger`: Sanger format (phred+33, 0 to 40)
- `illumina1.3`: Illumina 1.3+ format (phred+64, 0 to 40)
- `illumina1.5`: Illumina 1.5+ format (phred+64, 3 to 41)
- `illumina1.8`: Illumina 1.8+ format (phred+33, 0 to 41)

Default: `sanger`
Configuration keyword: `quality_encoding`

--list_file

When this command-line option is used, GeFaST considers its second argument as the path to a single plain-text file containing the list of actual input files. Each file path occupies a different line. Empty lines and comment lines (starting with #) are ignored.

Clustering and refinement options.

-t, --threshold *non-negative number*

Distance threshold used during the clustering phase to match amplicons. Two amplicons can only be considered similar when their distance is not larger than the specified value. The actual meaning of the threshold depends on the used mode and distance function.
Default: — (mode-dependent)
Configuration keyword: `threshold`

-r, --refinement_threshold *non-negative number(s)*

Distance threshold(s) used during the refinement phase. The argument to this option is either a single numerical value, a comma-separated list of values or a range. A range is specified by providing the first value, the last value (both inclusive) and an increment (all three separated by a colon). For example, the range 2:7:2 corresponds to the list 2,4,6. When specifying multiple refinement thresholds while using a non-iterative refiner, the largest specified value is used as the refinement threshold. Similar to the clustering phase, the actual meaning of the threshold(s) depends on the used mode and distance function.
Default: `0` (deactivated, no refinement)
Configuration keyword: `refinement_threshold`

-b, --boundary *positive integer*

Mass boundary used during the refinement phase to distinguish between light and heavy clusters. A cluster is considered light if and only if its mass is smaller than the specified boundary.
Default: `3`
Configuration keyword: `boundary`

-n, --break_swarms *binary*

Similar to Swarm, GeFaST can use the abundance values of amplicons to identify contacts between supposedly different clusters. When activated (value `1`), the abundance of similar amplicons is not allowed to be higher than the one of the current (sub)seed. Otherwise (value `0`), only the distance is considered when extending a cluster.
Default: `1`
Configuration keyword: `break_swarms`

-m, --match_reward *positive integer*

Reward for nucleotide match during pairwise global alignment computations.
Default: `5`
Configuration keyword: `match_reward`

-p, --mismatch_penalty *negative integer*

Penalty for nucleotide mismatch during pairwise global alignment computations.
Default: `-4`
Configuration keyword: `mismatch_penalty`

-g, --gap_opening_penalty *negative integer*

Penalty for opening a gap during pairwise global alignment computations.

Default: `-12`
Configuration keyword: `gap_opening_penalty`

-e, --gap_extension_penalty *negative integer*
Penalty for extending a gap during pairwise global alignment computations.
Default: `-4`
Configuration keyword: `gap_extension_penalty`

Component options. The following options can be used to, e.g., adapt `GeFaST` to the input type or to influence the fundamental behaviour during the clustering phase, by changing the major components controlling the different phases. The defaults are mode-dependent and not all modes allow changes through these options. More information on the restrictions are provided in the sections describing the mode-specific configurations and lists with the available implementations can be found below.

--preprocessor *string*
Use the specified component to perform the preprocessing (if possible).
Configuration keyword: `preprocessor`

--clusterer *string*
Use the specified component to cluster the amplicons (if possible).
Configuration keyword: `clusterer`

--refiner *string*
Use the specified component to refine the clusters (if possible).
Configuration keyword: `refiner`

--output_generator *string*
Use the specified component to generate the requested outputs (if possible).
Configuration keyword: `output_generator`

3.3 Major components

The following lists state the currently available implementations of the major components of `GeFaST`, provide the respective option value (in parentheses) and a outline their behaviour. A more thorough description of the different implementations is available in the `GeFaST` publications.

Preprocessors. By changing the preprocessor, `GeFaST` can handle different types of files but for a single run all files must have the same format. Some modes require particular input formats, while others accept different formats but keep only the information relevant for them.

- FASTA preprocessor (`fasta`): Supports multiline FASTA entries (i.e. the nucleotide sequence can be split over multiple lines). Additional information (after the first space) in the description lines of the entries are ignored.
- FASTQ preprocessor (`fastq`): Supports single-line FASTQ entries (i.e. the nucleotide sequence resp. quality scores must not be split over multiple lines). Additional information (after the first space) in the description lines of the entries are ignored. Dereplicates the sequences, combining the quality scores per position and summing the abundance. The way in which the quality scores are combined is adjustable through an option, which can be provided via the `misc` option:
  score_derep:*string*
  The quality scores of a dereplicated read are derived *per position* from the quality scores of the combined reads.
  – `mean`: Converts the quality scores into error probabilities, averages them and converts the average back into a quality score.
  – `max`: Selects the largest quality score.
  – `median`: Selects the median quality score.
  – `mode`: Selects the most common quality score.
  – `product`: Computes the product of the error probabilities corresponding to the quality scores. In order to stay within range of the used quality encoding, the error probability and, thus, the quality score are capped based on the highest quality score of the encoding.
  Default: `mean`

Clusterers. The mandatory clustering phase is the centrepiece of GeFaST as it performs the primary partitioning of the given amplicons into clusters. The respective methods are usually implemented in an abstract way that allows using different notions of distance (e.g. the number of edit operations in an optimal alignment or its score).

- Classic swarmer (`classic`): Implements the iterative clustering strategy of Swarm. Starting from a seed, amplicons are added iteratively to a cluster using a (small local) threshold. The cluster reaches its natural limit when no more amplicons can be connected to it via the latest generation of subseeds.
- Consistency-checked classic swarmer (`cons-classic`): Applies the same iterative strategy as `classic` and combines it with a consistency check inspired by the denoising method of DADA2. An amplicon is only added if it is similar to and consistent with the subseed. Requires quality information (e.g. from FASTQ files). The clusterer offers additional, component-specific options (borrowed from DADA2), which can be provided via the `misc` option:

  omega_a:*positive float*

  Abundance threshold for deciding the consistency. An abundance p-value below the threshold indicates that the new member would be significantly overabundant and, thus, not consistent.
  Default: `1e-40`

  error_matrix:*string*

  Path to file containing an alternative error matrix. The matrix contains a row for each pair of nucleotides and a column for each possible quality score. Each line of the file corresponds to one row of the matrix, with columns separated by a single space. If the file does not specify enough columns (for the used quality encoding), the last one is repeated. The default matrix distinguishes only between match and mismatch (instead of considering the actual nucleotides).
  Default: ϵ (empty string, use default matrix)
- Consistency-based swarmer (`cons-swarmer`): Applies the same iterative strategy as `classic` but uses only the consistency check inspired by DADA2. An amplicon is added if it is consistent with the subseed. Requires quality information (e.g. from FASTQ files). The clusterer offers the same additional options as `cons-classic`.
- Dereplicator (`dereplicator`): Cluster the amplicons using exact sequence equality.

Cluster refiners. GeFaST offers an optional refinement phase, which aims for improving the quality of the clusters obtained in the previous clustering phase. Similar to Swarm, GeFaST distinguishes clusters based on their mass and tries to graft light clusters onto heavy ones.

- Classic fastidious refiner (`classic`): Implements a generalised version of the fastidious refinement proposed in Swarm. By using an increased (originally doubled) threshold, additional links between amplicons from light and heavy clusters are searched. In contrast to Swarm, this refinement can be used irrespective of the clustering threshold and the refinement threshold can be freely chosen.
- Iterative fastidious refiner (`iterative`): Performs several rounds of the fastidious refinement with increasing refinement threshold. The individual rounds behave like the `classic` refiner, except that light clusters already attached in a previous round are not considered again.
- Light-swarm appender (`cons-lsa`): Searches for consistencies between the seeds of light and heavy clusters. A light cluster is grafted as a whole if such a consistency is found. When the light cluster is consistent with multiple heavy clusters, the one with the highest expected abundance is chosen. Requires quality information (e.g. from FASTQ files). The refiner offers additional, component-specific options, which can be provided via the `misc` option:

  omega_a:*positive float*

  Abundance threshold for deciding the consistency. An abundance p-value below the threshold indicates that the new member would be significantly overabundant and, thus, not consistent.
  Default: `1e-40`

  error_matrix:*string*

  Path to file containing an alternative error matrix. The matrix contains a row for each pair of nucleotides and a column for each possible quality score. Each line of the file corresponds to one row of the matrix, with columns separated by a single space. If the file does not specify enough columns (for the used quality encoding), the last one is repeated. The default matrix distinguishes only between match and mismatch (instead of considering the actual nucleotides).
  Default: ϵ (empty string, use default matrix)

light_opt:*integer*

Processing option for light clusters still unattached after the refinement. Currently, the following options are available: (`1`) keep the clusters without further changes, (`2`) discard them, (`3`) combine them into a single (star-shaped) cluster, or (`4`) combine and repartition them using DADA2's split-and-shuffle strategy.
Default: `1`

- Light-swarm resolver (`cons-lsr`): Disassembles the light clusters and searches for consistencies between the individual amplicons obtained from the light clusters and the seeds of heavy clusters. An amplicon is grafted if such a consistency is found, choosing the target cluster based on the highest expected abundance when there are multiple ones. Requires quality information (e.g. from FASTQ files). The refiner offers the same additional options as `cons-lsa`.

- Light-swarm shuffler (`cons-lss`): Searches for consistencies between the individual amplicons contained in the light clusters and the seeds of heavy clusters. The amplicons can leave their cluster individually when a consistency is found, again choosing the target cluster based on the highest expected abundance when there are multiple ones. The seed of the light cluster can only be grafted if it is the last amplicon remaining in its cluster. When not all amplicons of a light cluster are grafted, it is rearranged into a star-shaped cluster with the seed as the centre. Requires quality information (e.g. from FASTQ files). The clusterer offers the same additional options as `cons-lsa` (except for `light_opt`).

Output generators. Currently, the output generator is determined by the mode and cannot be changed by the user.

- Classic output generator (`classic`): Mimics the general output behaviour of `Swarm`. The clusters are ordered by the abundance of the seed.

- Dereplication output generator (`dereplication`): Mimics the output behaviour of `Swarm` for `-d 0` (dereplication). The clusters are ordered by their mass.

4 Mode-specific configuration

The following sections describe mode-specific parts of the configuration. These can involve additional options, default values and lists of selectable components. Mode-specific options have to be provided via the configuration file (some via the **misc** option).

4.1 Levenshtein mode (lev)

The Levenshtein mode can use the following components (defaults are marked with *):

- Preprocessor: `fasta`*, `fastq`
- Clusterer: `classic`*, `cons-classic`
- Refiner: `classic`*, `iterative`, `cons-lsa`, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

The default value of the clustering `threshold` is 1.

Additional options:

num_extra_segments=*positive integer*
Change the number of extra segments used by the segment filter trying to avoid unnecessary alignment computations.
Default: `1`

two_way_segment_filter=*binary*
When set to `1`, a bidirectional segment filter is used, adding a pipelined second filtering step in order to increase the filtering capacity of the segment filter.
Default: `0`

use_score=*binary*
When set to `1`, the distance between two amplicons is computed as the number of edit operations in an optimal alignment according to the given scoring function (see the corresponding options in Section 3.2). Otherwise, the actual Levenshtein (edit) distance is computed.
Default: `0`

use_qgrams=*binary*
When set to `1`, a lower bound on the q-gram distance between two amplicons is computed to avoid unnecessary alignments.
Default: `0`

4.2 Alignment-score mode (as)

The alignment-score mode can use the following components (defaults are marked with *):

- Preprocessor: `fasta`*, `fastq`
- Clusterer: `classic`*, `cons-classic`
- Refiner: `classic`*, `iterative`, `cons-lsa`, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

The default value of the clustering `threshold` is 40.

Additional options:

bands_per_side=*non-negative integer*
Limit the alignment computation to a number of bands on either side of the main diagonal. Speeds up the computation but might exclude the optimal alignment when chosen too small.
Default: `-1` (automatic computation based on threshold and scoring function)

num_extra_segments=*positive integer*
Use a segment filter with the specified number of extra segments to avoid unnecessary alignment computations.
Default: `0` (segment filter not used)

use_qgrams=*binary*

When set to `1`, a lower bound on the q-gram distance between two amplicons is computed to avoid unnecessary alignments.
Default: `0`

4.3 Alignment-free mode (af)

The alignment-free mode can use the following components (defaults are marked with *):

- Preprocessor: `fasta`*, `fastq`
- Clusterer: `classic`*, `cons-classic`
- Refiner: `classic`*, `iterative`, `cons-lsa`, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

There is no default clustering `threshold` and it has to be specified by the user, because the different feature-vector representations and distance functions tend to work with very different thresholds. The table at the end of the section provides starting points for parameter and threshold selection.

Additional options:

distance=*string*

Distance function used to compute the distance between the feature vectors representing the amplicon sequences. `GeFaST` offers the following distance functions:

- `manhattan`: Manhattan distance
- `euclidean`: Euclidean distance
- `cosine`: Cosine distance
- `pearson`: Pearson distance

Some distance functions can be combined with auxiliary data structures (see below).
Default: (no default value, has to be specified explicitly)

representation=*string*

Feature-vector representation used to compare sequences (instead of computing an alignment between them). Currently, the following representations are available:

- `wcv`: Represents a sequence by a s^k-dimensional word-composition vector (for s the size of the alphabet) counting how often each possible k-mer occurs in it. Works with arbitrary alphabets.
- `cpf`: Represents a sequence by a 12-dimensional vector containing entropy values based on the occurrences and chemical characterisations of dinucleotides. Works only with the alphabets `ACGT` and `ACGU`. Based on work by Bao et al. (doi: 10.1186/1471-2105-15-321).
- `det`: Represents a sequence by a s^2-dimensional vector (for s the size of the alphabet) describing the distance-weighted relationships between the alphabet symbols. Works with arbitrary alphabets. Based on work by Qi et al. (doi: 10.4137/EBO.S7364).
- `bbc`: Represents a sequence by a s^2-dimensional vector (for s the size of the alphabet) describing the mutual information of the different alphabet symbols up to a maximum distance. Works with arbitrary alphabets. Based on work by Liu et al. (doi: 10.1016/j.bbrc.2008.01.070).
- `2d-w`: Represents a sequence by two-dimensional vector describing the geometric centre of the two-dimensional walk induced by the sequence. Works only with the alphabets `ACGT` and `ACGU`. Based on work by Liao (doi: 10.1016/j.cplett.2004.11.059).
- `cgr`: Represents a sequence by a three-dimensional vector describing the geometric centre and mean deviation of the classic CGR (Chaos Game Representation) of the sequence. Works only with the alphabets `ACGT` and `ACGU`. Based on work by Almeida et al. (doi: 10.1093/bioinformatics/17.5.429).
- `mcgr`: Represents a sequence by a nine-dimensional vector describing the geometric centres and mean deviations of multiple CGRs of the sequence, considering the chemical properties of the nucleotides. Works only with the alphabets `ACGT` and `ACGU`. Based on work by Deng & Luan (doi: 10.1155/2013/926519).
- `3d-cgr`: Represents a sequence by a 16-dimensional vector describing the geometric centres and mean deviations of four three-dimensional CGRs of the sequence. Works only with the alphabets `ACGT` and `ACGU`. Based on work by Huang & Shi (doi: 10.1109/BMEI.2010.5639720).

Parameters specific to the different representations can be provided via the `misc`-option (see below).
Default: (no default value, has to be specified explicitly)

auxiliary=*string*
Auxiliary data structures improving the runtime. By default, all pairwise comparisons (ignoring only the amplicons already added to some swarm) are performed. The Manhattan and Euclidean distance can be combined with a space-partitioning data structure (k-d tree) to reduce the number of comparisons (option value: `kdtree`).
Default: `naive`

Additional misc*-based options for the feature-vector representations*:

wcv_word_length:*positive integer*
The length of the words (k-mers) counted. Applies only to `wcv`.
Default: (no default value, has to be specified explicitly)

det_alpha:*positive float*
The decay rate α of the decreasing function $d^{-\alpha}$ used to weight the relation between two nucleotides d positions apart. `det_alpha` and `det_pref_dist` cannot be used at the same time. Applies only to `det`.
Default: (no default value, has to be specified explicitly)

det_pref_dist:*positive integer*
Alternative way to compute α based on the preference distance, i.e. the largest distance d with $d \geq 0.1$. `det_pref_dist` and `det_alpha` cannot be used at the same time. Applies only to `det`.
Default: (no default value, has to be specified explicitly)

bbc_max_dist:*positive integer*
Restricts the distance up to which the relations between nucleotides are considered. Applies only to `bbc`.
Default: (no default value, has to be specified explicitly)

2dw_alpha:*float*
Incremental parameter α used to transform the amplicon sequence into a walk. Applies only to `2d-w`.
Default: (no default value, has to be specified explicitly)

2dw_beta:*positive float*
Incremental parameter β used to transform the amplicon sequence into a walk. Should be chosen such that $\sqrt{\beta} \notin \mathbb{Q}$. Applies only to `2d-w`.
Default: (no default value, has to be specified explicitly)

As mentioned above, the different feature representations tend to work with very different thresholds. Furthermore, they are not as intuitive as, e.g., the number of differences in an alignment. Preliminary studies on the data sets used to evaluate the alignment-based and quality-aware methods gave a first impression on suitable parameters and thresholds. The following table provides initial threshold values (roughly corresponding to a threshold of 1 for the Levenshtein distance) and step sizes (roughly corresponding to increases of 1 in terms of the Levenshtein distance). These values together with the selected parameters should only be considered as mere starting points because their broader applicability remains to be explored.

Representation	Parameter values	Manhattan		Euclidean	
		Initial	Step	Initial	Step
`wcv`	4	9.0	5.2	3.2	0.6
`cpf`	—	0.06	0.02	0.03	0.01
`det`	20	32.7	8.4	10.8	2.5
`bbc`	40	0.22	0.05	0.07	0.01
`2d-w`	$\alpha = \frac{1}{4}, \beta = \frac{3}{4}$	0.69	0.19	0.52	0.16
`cgr`	—	0.008	0.002	0.006	0.001
`mcgr`	—	0.022	0.005	0.010	0.002
`3d-cgr`	—	0.042	0.009	0.013	0.003

4.4 Quality Levenshtein mode (qlev)

The quality Levenshtein mode can use the following components (defaults are marked with *):

- Preprocessor: `fastq`*
- Clusterer: `classic`*, `cons-classic`
- Refiner: `classic`*, `iterative`, `cons-lsa`, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

The default value of the clustering `threshold` is 1.

Additional options:

num_extra_segments=*positive integer*
Change the number of extra segments used by the segment filter trying to avoid unnecessary alignment computations.
Default: `1`

two_way_segment_filter=*binary*
When set to `1`, a bidirectional segment filter is used, adding a pipelined second filtering step in order to increase the filtering capacity of the segment filter.
Default: `0`

use_qgrams=*binary*
When set to `1`, a lower bound on the q-gram distance between two amplicons is computed to avoid unnecessary alignments.
Default: `0`

distance=*string*
The different quality-weighted alignment methods have been implemented as the following distance functions for the quality Levenshtein mode:

- `clement`: Score a (mis)match by the linear combination of all possible substitutions, weighted by probabilities. Based on work by Clement et al. (doi: 10.1093/bioinformatics/btp614).
- `converge-a`: Let the costs of the operations converge with decreasing quality against a balance cost positioned between the match reward and the penalties of the other edit operations.
- `converge-b`: Let the costs of the operations converge with decreasing quality against each other, up to an operation-specific amount.
- `frith`: Modify the provided scoring matrix (derived from the scoring function) by incorporating error probabilities into the underlying likelihood ratios. Based on work by Frith et al. (doi: 10.1093/nar/gkq010).
- `kim-a`: Score an operation by the linear combination of all possible substitutions, insertions and deletions, weighted by approximated probabilities disregarding contributions expected to be small. Based on work by Kim et al. (doi: 10.1016/j.pnsc.2007.12.011).
- `kim-b`: Score an operation by the linear combination of all possible substitutions, insertions and deletions, weighted by precise probabilities. Based on the same work as `kim-a`.
- `malde-a`: Compute an alternative scoring matrix by only using combined error probabilities. Based on work by Malde (doi: 10.1093/bioinformatics/btn052).
- `malde-b`: Weight substitution costs directly by the combined error probabilities. Based on the same work as `malde-a`.
- `malde-c`: Weight substitution, insertion and deletion costs directly by the (combined) error probabilities. Based on the same work as `malde-a`.

Some of them have specific parameters, which can be provided via the `misc`-option (see below).
Default: `frith`

Additional misc*-based options*:

unweighted_matches:*bool*
When set to `true`, match operations in the alignments are not affected by the quality-weighting mechanism. As a result, gradually accumulating contributions from (even high-quality) matches that could lead to an undesirable or unexpected dependency between distance threshold and sequence length are avoided.
Default: `false`

inner_boost:*bool*

When set to `true`, the individual error probabilities of the nucleotides are boosted. Otherwise, the boosting function is applied to combined probabilities associated with observing an operation.
Default: `true`

boosting_method:*string*

Boosting functions can be used to emphasise the differences between the quality scores despite the exponential decay of the associated error probabilities. Currently, the following boosting functions are available:

- `none`: The probabilities are not boosted.
- `linear`: This boosting function keeps the exponential decay up to a specified quality score and switches to a linear decay for higher scores.
- `mult`: The probabilities are multiplied by the specified factor (with the product being capped at 1).
- `root`: This boosting function extracts a root of the probabilities and, optionally, shifts the result towards zero. Both the degree and the shift can be specified.

Parameters specific to the different boosting functions can be provided via the `misc`-option (see below).
Default: `none`

Additional misc*-based options for the distance functions*:

balance_factor:*float*

Influences the point of convergence of the costs (the lower the value, the closer to the match costs) . The value should be chosen between 0 and 1. Applies only to `converge-a`.
Default: `0.5`

max_change_match:*float*

The convergence of the match costs is limited by the specified maximum deviation. Applies only to `converge-b`.
Default: half the transformed mismatch costs

max_change_mismatch:*float*

The convergence of the mismatch costs is limited by the specified maximum deviation. Applies only to `converge-b`.
Default: half the transformed mismatch costs

max_change_gap_open:*float*

The convergence of the gap-opening costs is limited by the specified maximum deviation. Applies only to `converge-b`.
Default: half the transformed gap-opening costs

max_change_gap_extend:*float*

The convergence of the gap-extension costs is limited by the specified maximum deviation. Applies only to `converge-b`.
Default: half the transformed gap-extension costs

scaling_factor:*float*

Scaling factor of the scoring-matrix computation. Applies only to `frith` and `malde-a`.
Default: 5.22113 (`frith`) resp. $\frac{1}{\ln(2)}$ (`malde-a`)

Additional misc*-based options for the boosting functions*:

linear_start:*integer*

The quality level (score) at which the linear decay starts. Applies only to `linear`.
Default: `0`

linear_levels:*integer*

The number of levels in the used quality encoding. Applies only to `linear`.
Default: automatically determined from the quality encoding

mult_factor:*float*

The constant factor of the multiplicative reinforcement. Applies only to `mult`.
Default: `1.0`

root_degree:*positive integer*
: The degree of the root to be extracted. Applies only to `root`.
Default: `2`

root_shift:*float* or *string*
: After extracting the root, the result is shifted towards 0 by the specified amount. When specifying the value `full`, the shift value is determined based on the quality encoding in a way that the boosted error probability for the highest quality score becomes 0. Applies only to `root`.
Default: `0`

4.5 Quality alignment-score mode (qas)

The quality alignment-score mode can use the following components (defaults are marked with *):

- Preprocessor: `fastq`*
- Clusterer: `classic`*, `cons-classic`
- Refiner: `classic`*, `iterative`, `cons-lsa`, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

The default value of the clustering `threshold` is 40.

Additional options:

bands_per_side=*non-negative integer*
: Limit the alignment computation to a number of bands on either side of the main diagonal. Speeds up the computation but might exclude the optimal alignment when chosen too small.
Default: `-1` (automatic computation based on threshold and scoring function)

num_extra_segments=*positive integer*
: Use a segment filter with the specified number of extra segments to avoid unnecessary alignment computations.
Default: `0` (segment filter not used)

use_qgrams=*binary*
: When set to `1`, a lower bound on the q-gram distance between two amplicons is computed to avoid unnecessary alignments.
Default: `0`

distance=*string*
: The nine quality-weighted alignment methods listed for the quality Levenshtein mode have also been implemented as distance functions for the quality alignment-score mode. However, the values to select the different methods are prefixed with `score-`, e.g. leading to `score-clement` instead of `clement`. In this case, the alignment computations are not banded (as the quality-weighted operation costs might make it necessary to consider all diagonals). A banded computation (by default estimating the number of bands from the unweighted costs) can still be activated by using the prefix `banded-score-` instead. The parameters specific to the different methods are again provided via the `misc`-option.
Default: `banded-score-frith`

The additional *misc*-options (also for the distance functions and boosting functions) are the same as for the quality Levenshtein mode.

4.6 Consistency mode (cons)

The consistency mode can use the following components (defaults are marked with *):

- Preprocessor: `fastq`*
- Clusterer: `cons-swarmer`*
- Refiner: `cons-lsa`*, `cons-lsr`, `cons-lss`
- Output generator: `classic`*

The mode has no mode-specific options (beyond the component-specific options related to the consistency).

4.7 Dereplication mode (derep)

The dereplication mode can use the following components (defaults are marked with *):

- Preprocessor: `fasta`*, `fastq`
- Clusterer: `dereplicator`*
- Refiner: `idle`* (refinement is never used)
- Output generator: `dereplication`*

The mode has no mode-specific options.