Contents

CONTENTS

1 | Introduction

This manual is written as a guide for scientists using and continuing to develop *MNHN-Tree-Tools*. As such this guide provides you with a installation procedure, detailed usage instructions and technical details for every tool contained within this suite and a tutorial to get the reader started with a typical clustering and density distance based phylogenetics as well as tree building.

1.1 License

1.2 General Overview

MNHN-Tree-Tools was developed in order to cluster and infer phylogenetic evolution into repeated sequences in a single genome, namely: α-satellites in primate genomes. During the development the versatility of the tools presented herein was both discovered and enforced. As such this tool is applicable to all kinds of data discovery tasks on datasets of sequences. Notably this tool has been used for experiences around Deoxyribonucleic Acid (DNA) barcoding tasks Operational Taxonomic Unit (OTU) detection.

MNHN-Tree-Tools is a suite that allows you to gain insights into a dataset composed of nucleic or protein sequences provided as FASTA [1] files. The tool has the ability to cluster such a dataset and to built phylogenetic trees based on a density-distance based criteria. This is achieved by a repeated application of the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [2] algorithm. Clustering and gaining such sequence-sequence dependency or phylogentic insights are at the heart of this software. Nevertheless this software provides you with all kinds of tools that might be of use, such as, but not limited to: The generation of fasta datasets using simulated evolution, the projection of FASTA files into a Principal Component Analyses (PCA) based subspace, the evaluation of k-mer vectors from fasta datasets or the calculation of clustering parameters such as the Silhouette [3] index on a dataset.

2 | Installation

2.1 Generic Installation

In this chapter we will provide you with the details that allow you to install *MNHN-Tree-Tools*. The primary distribution method is in the form of source code on available from:
`https://gitlab.in2p3.fr/mnhn-tools/mnhn-tree-tools`

For a full installation a machine with a UNIX or UNIX-like system is required. The machine currently has to be of *x86_64* architecture and support the *AVX* instruction set. In order to compile the full suite of tools you need a toolchain, consisting of:

- *make:* Which is required to build the suite and various subsets
- *compiler/linker:* A working C compiler, the suite was found to compile with *gcc* and *clang*.
- *BLAS/LAPACK* An implementation of the BLAS/LAPACK libraries.
- *OpenCL:* A working implementation of the OpenCL language. The suite was tested with the *nvidia-opencl-toolkit* and the *intel-opencl-sdk*.
- *SDL2:* An implementation of the SDL2 library.
- *MPI:* An implementation of the Message Passing Interface (MPI). The suite was tested with *OpenMPI*.
- *PNG:* An implementation of the PNG graphics library.

You may make sure that you not only have the binary versions of the supporting libraries installed but also the header files as the software will fail to compile otherwise. As such, depending on your software distribution and operating system, you may need to install the developer *-dev* versions of the libraries listed above. The *MNHN-Tree-Tools* source code is shared using git and may

be obtained by issuing the following command in your UNIX terminal:

```
git clone https://gitlab.in2p3.fr/mnhn-tools/mnhn-tree-tools.git
cd mnhn-tree-tools
```

Listing 2.1 – Cloning the source from github.

As you have cloned and entered the mnhn-tree-tools directory you need to edit the head of the make file to suite your UNIX systems needs, and set the correct compiler flags and link commands. We refer to the documentation and formus accompanying your operating system to find the correct values. In general *CC* reflects the command of your compiler, *MPICC* the MPI wrapper of your compiler and *CFLAGS* the options that you want to pass to your compiler. The other options correspond to the flags required by the linker to find the appropriate libraries.

```
CC=gcc
MPICC=mpicc
CFLAGS= -O2 -march=native -ftree-vectorize -fomit-frame-pointer
LAPACK=-llapack
MATH=-lm
PTHREAD=-pthread
OPENCL=-lOpenCL
PNG=-lpng
MPI=-lmpi
SDL=$(shell sdl2-config --cflags) $(shell sdl2-config --libs)
```

Listing 2.2 – Head of the Makefile

With your flags adjusted you may compile the code issuing

```
mkdir bin
make all
```

Listing 2.3 – Building all tools

If you do not need the full functionality of *MNHN-Tree-Tools* you can install each tool individually by calling:

```
mkdir bin
make name-of-the-tool
```

Listing 2.4 – Building individual tools

This might especially be convenient if you are not able to install all the libraries in question on your system as the tool that you might really need might not depend on the library that you might not be able to install.

2.2 Quick Installation on Debian 10 / Ubuntu 20.04

If you just want to get all tools working quickly, we provide you with the following quick install script for *Debian 10* and derived distributions:

```
sudo apt-get install git build-essential libpng-dev libsdl2-dev \
 liblapack-dev libopenmpi-dev libpocl-dev ocl-icd-opencl-dev pocl-opencl-icd

git clone https://gitlab.in2p3.fr/mnhn-tools/mnhn-tree-tools.git

cd mnhn-tree-tools
mkdir bin
make all
cd bin

# make MNHN-TREE-TOOLS available from any folder ( this is temporary, you
# may modify your .bashrc and similar files to make this permanent
export PATH=$PATH:$PWD
```

Listing 2.5 – Installing MNHN-Tree-Tools on Debian 10

For optimal performance we recommend you to use a more elaborate installation as shown in section 2.1 as this allows to run OpenCL code on the GPU and the use of optimized math (Lapack) and MPI libraries that might be available on your system.

3 | Tutorial

In this chapter we will walk you through basic manipulations of nucleic datasets, tree building, naming conventions and overall usage to get you started with *MNHN-Tree-Tools*.

3.1 Preparing your data

At first let us get started with a dataset that you might have. For this tutorial we recommend you to have a dataset of reasonable size at hand. While these tools work and have been tested for millions of sequences we recommend you that you use a dataset of about 5000 to 10000 sequences for this tutorial to progress fast and learn how to use these tools quickly. If you have no dataset available you can generate one with our built in virtual biosphere generator, the *virtual_evolution* tools shown in section 4.24. To generate such a dataset you might run:

```
virtual_evolution_controlled  15 100 10000 2 -1 50000 5 test.fasta
```

Listing 3.1 – Generate a test dataset using the *virtual_evolution* tools

which should yield a decent dataset to discover *MNHN-Tree-Tools*.

The tools presented herein are built with FASTA [1] files in mind. If you have your sequence data available in a different file format you may have to convert it beforehand. If you bring your own FASTA file for this tutorial please note that the *MNHN-Tree-Tools* are only able to treat human readable non single line FASTA files. As such a sequence may not contain more than 1000 characters per line. If your file contains lines that are longer then the 1000 character limit you may convert it to a conforming FASTA file using the *seqret* tool from EMBOSS [4]. Several application domains of *MNHN-Tree-Tools* are used to use FASTQ files. If

you have datasets in this format you may use the following *bash* script to convert your FASTQ files to FASTA files.

```
#!/bin/bash

sed -n '1~4s/^@/>/p;2~4p' $1 > $2
```

Listing 3.2 – Script to convert FASTQ to FASTA files

In such a case it might again be necessary to apply *seqret* from EMBOSS on the FASTA files resulting from this conversion in order to ensure appropriate line lengths.

Before using a dataset with *MNHN-Tree-Tools* you further may make sure that it is suited for the task that you would like to achieve. The tools presented herein are in general made to compare sequences of at least a reasonable similarity. As calculating the consensus of unaligned datasets consisting of millions of sequences might be cumbersome, we suggest to examine the variation in sequence lengths first. The *lengths_from_fasta* tool is perfect for this task (c.f. section 4.10) and may be run like:

```
lengths_from_fasta your.fasta > lengths
```

Listing 3.3 – *lengths_from_fasta* tutorial

The tool yields all the sequence lengths for the sequences of your dataset on separated lines in the *lengths* file. We leave it to the reader to analyze with the divers tools available to find out weather the lengths do have a variance that is reasonable in respect to the average sequence length.

3.2 Investigating k-mers and Principal Components

A convenient way to transform a set of sequences into an alignment free representation, hence a representation where we do not need to align sequences in order to calculate their distance or compare them is the *k-mer* representation. In the *k-mer* representation we compare all possibilities of sequences of certain k length with a sequence in questions and count how often such a sequence appears in our sequence of question. Let us imagine that k equals four. In such a case we obtain 256 possibilities to form a sequence (out of nucleotides) or 4^4 sequences. As we count how often any of these sequences occurs in our target sequence, we can represent the target sequence as a vector

of 256 k-mer frequencies, a vector holding the counts how often each sequence occurs in the target sequence for the 256 k-mers. In order to perform this transformation we run the *fasta2kmer* tool (c.f. section 4.1). In our case we are not generating 4-mers, but 5-mers for our sequences and thus vectors of $4^5 = 1024$ values in length:

```
fasta2kmer test.fasta 5 2 0 > test.kmer
```

Listing 3.4 – Generating 5-mers with *fasta2kmer*

on a supposedly dual core machine, and hence we are selecting 2 threads. In order to choose the right *k-mer* size you may be advised to use a length that captures all the information in your sequences. Hence the longer your sequences are, the higher the k of the sequences shall be. At the same time you shall be aware that every increase in k also generates an increase of 4^k data generated and therefore we strongly discourage using $k > 7$ if the length of your sequences permits such a choice. To encode enough information we suggeste that you have about as many k-mers at hand as the average number of nucleotides contained in a sequence in your dataset. Thus armed with a k-mer length of $k = 4$, for instance we shall be fine for sequences of up to 256 nucleotides. Typing

```
head -n 2 test.kmer
```

Listing 3.5 – Investigating 5-mers with head

allows us to investigate the first two lines of our calculated *k-mers* file and as such the representation in k-mer form of the first two sequences of a supplied FASTA dataset *test.fasta*. Having the k-mer representation of our sequences at hand we can perform a dimensionality reduction step, selecting a representation in a subspace that still covers the variance of dataset, a method called principal component analysis (PCA). For in depth details we refer you to your favorite *linear algebra* textbook. To perform PCA on the dataset we have the *kmer2pca* tool (c.f. section 4.2) at hand. Applying the tool on our *test.fasta* dataset together with its k-mer representation *test.kmer*, we are able to perform a PCA by issuing the following command:

```
kmer2pca test.kmer test.pca test.ev 7 2
```

Listing 3.6 – Performing PCA on the k-mer representation of our sequences

and are using our supposed dual core machine, calculating the projections onto the seven first principal components with the largest corresponding eigenvalues. The eigenvalues printed in test.ev allow us to get an idea of the information that we capture in our 7 dimensional PCA subspace. A good measure is to perform the sum of all eigenvalues, and then to sum up, in this case the first 7 and calculate the percentage of that the sum of the first seven represents over the sum of all eigenvalues. Even if this value might seem low, subspaces with a dimensionality that is larger than 7 can lead to the *curse of dimensionality* in the subsequent clustering parts of this tutorial, and we might note that even with a capture below 30% our algorithm still yielded interesting results.

But first let us investigate the PCA subspace, and the sequences in it. In order to do this we can use the *gnuplot* program [5]: In the gnuplot shell one can call:

```
plot 'test.pca' using 1:2
plot 'test.pca' using 1:3
splot 'test.pca' using 1:2:3
```

Listing 3.7 – Plotting PCA with *gnuplot*

where the *1:2* represents the first and second component. As our subspace is 7 dimensional we can plot 2 or 3 dimensional projections. The principal components represent our axes and as it is projection and rotation the units are *k-mer* frequencies. Generating images like the ones shown in figure 3.1: As outlined by the figure it might not be evident to detect different clusters (in this case 2 as the projection shows a dataset like the one generated with the command shown in listing 3.1). For a deeper inspection you might refer to the *pca2densityfile* and *pca2densitymap* tools shown in sections 4.27 and 4.26 respectivly. Besides this tool we further have a PCA sequence selector, that allows us to extract and investigate sequences, from dense ensembles, in literature also cited as comets [6], and directly access the sequences. This tool is highlighted in section 4.3.

3.3 Adaptive Clustering

One core feature of *MNHN-Tree-Tools* is an adaptive density based clustering algorithm that can be used to study phylogenetic features and the evolution of species by investigating typ-

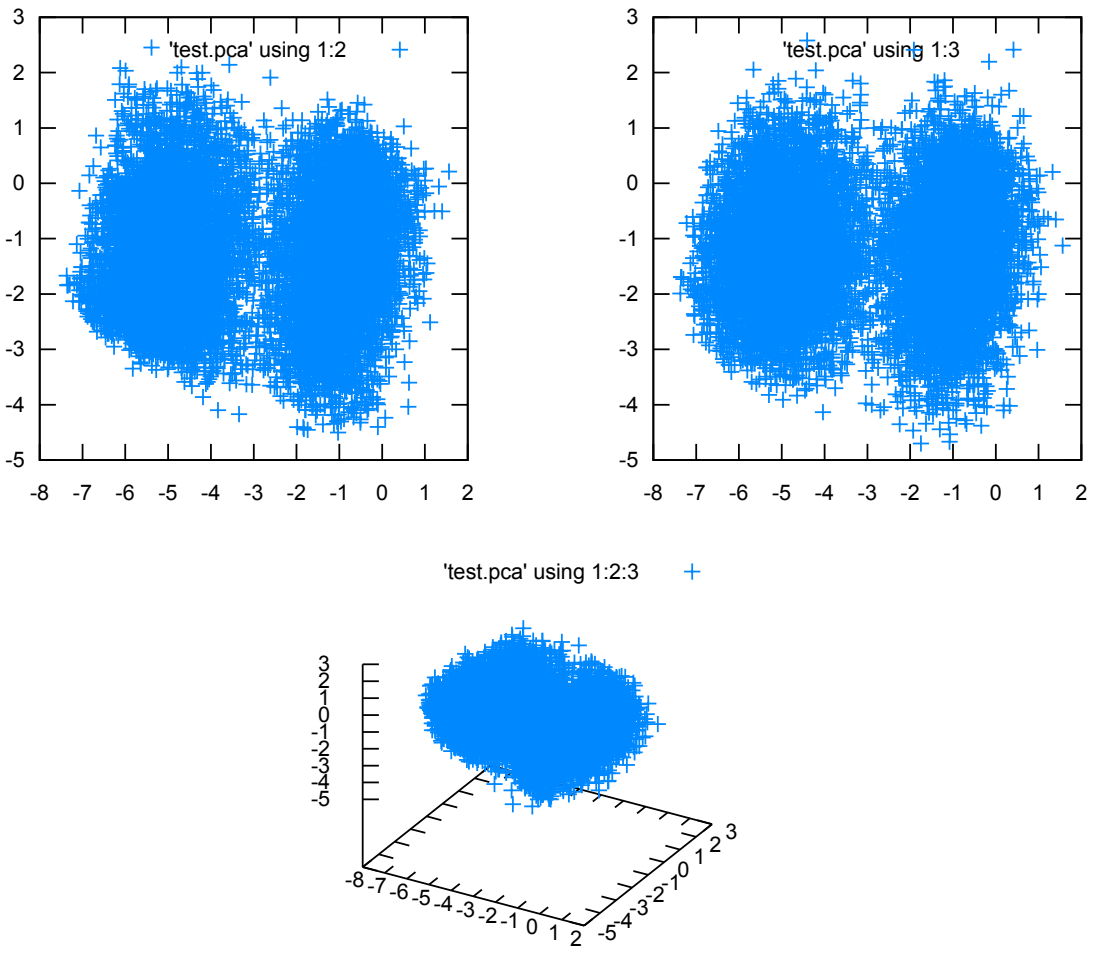


Figure 3.1 – Example plots of PCA projections, upper left composants 1 and 2, upper right 1 and 3 and lower 1,2 and 3

ical DNA barcoding targets such as the 16S or 18S ribosomal RNA or Cyclooxygenase 1 (COX1) coding regions. Further the tool might be of use for all kinds of purposes where similar sequence clusters are to be investigated. As such *MNHN-TreeTools* was successfully applied in the quest to discover families of α-satellites, a set of repeated sequences found in the human genome.

Adaptive clustering is implemented in the MNHN tools with different underlying sequence-distance measures and optimized for different computing architectures. As we have already highlighted the aforementioned PCA tools during and also for the sake of rapidity we use the PCA based adaptive clustering method in order to investigate our dataset of sequences further:

```
mkdir clusters
adaptive_clustering_PCA test.fasta 1 0.001 3 clusters/L 2 7 test.pca > cl-out
```

Listing 3.8 – Performing PCA based adaptive clustering

By issuing this command we iteratively use the DBSCAN algorithm [2] from a starting value of $\epsilon_{\text{int}} = 1$ and increasing in it each step by $\Delta\epsilon = 0.001$. Together with the constant *minPoints* value 3 this forms different density limits:

$$\rho_{\text{clust}} > \rho_{\text{limit}}(\epsilon, \text{minpts}) = \frac{\text{minpts}}{V(\epsilon)} \tag{3.1}$$

The scanning algorithm will find us sequences above these density limits, and as the density decreases in each step regions that house connected sequences above a certain density are growing and will merge, and hence, denser regions are found to be in less denser regions. These denser regions in less denser regions allow us the interpret dependencies, build dendograms and investigate the evolutionary history of the sequences. A simple illustration of the algorithm is shown in figure 3.2. For a more detailed overview the reader is referred to section 4.12. The command in listing 3.8 yields us with an output file *cl-out* and a number of of binary *split-sets* in the *cluster* folder. A binary *split-set* is a file that holds a partitioning of our dataset. As we have run an adaptive clustering run, several different partitions for different densities have been returned by the algorithm. These will later form the layers of our tree. As such we named them L. The tool returned us files named like *L0000*, *L0001* etc. indicating the layer of the tree from the most outer leaf node *L0000* to

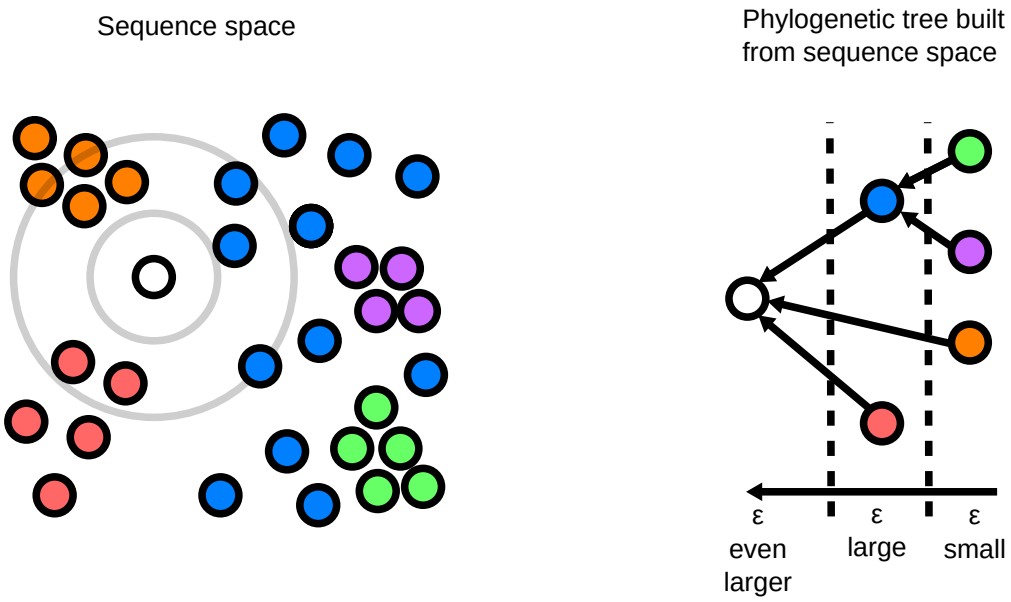


Figure 3.2 – The adaptive clustering algorithm finds dense (new) clusters in less dense (old) more dispersed clusters. Finally the *MNHN-Tree-Tools* collection allows us to build a tree like the one on the right, from a sequence space as shown on the left.

the most inner root *LNNNN* with *NNNN* being the highest number available. For each of the clusterings we also know, thanks to the data that we stored in *cl-out* which binary *split-set* belongs to which ϵ and hence, to which density limit.

3.4 Tree Building, Tree Visualization and Tree Investigation

With our *split-sets* from an adaptive clustering run at hand, and having looked already at the *cl-out* file one might already guess the next step in this tutorial. The *cl-out* file contains a list of connections between the different clusters in the different layers. The next step is to build a tree, or dendogram, representing the density structure of our dataset and hence, obtain the right side of figure 3.2. *MNHN-Tree-Tools* provides a tool to infer a Newick tree [7] from the clusters obtained during an adaptive clustering run. Here we run the tool *split_sets_to_newick* outlined in section 4.15 using the results from our adaptive clustering run:

```
split_sets_to_newick 0 0 clusters/L* > tree.dnd
```

Listing 3.9 – Generating a Newick tree with *split_sets_to_newick*

and obtain a Newick tree in the file *tree.dnd*. You can use any tool of your choice to plot the Newick tree. Herein we prefer the Newick tools by the University of Geneva [8].

```
nw_display -s -b 'opacity:0' -w 600 tree.dnd > tree.svg
```

Listing 3.10 – Using the Newick tools [8] to plot a dendogram

where the options are: *-s* in order to generate a *scalable vector graphics* (SVG) file, *-b 'opacity:0'* is here to hide the number indicating the distance between the clusters. *-w* the size, with of the image. The reader shall be referred to the Newick tools manual for all the vast possibilities this tool provides. If you use your own dataset you might have to adjust the width so that all the labels are visible. In general you should be presented with a tree like the one shown in figure 3.3. The image shown in figure 3.3 outlines the tree of clusters obtained from our virtual biosphere dataset that we generated issuing the command in listing 3.1 where two partitions with a distance of at least 5 nucleotides was requested. One of these underwent 10000 single nucleotide

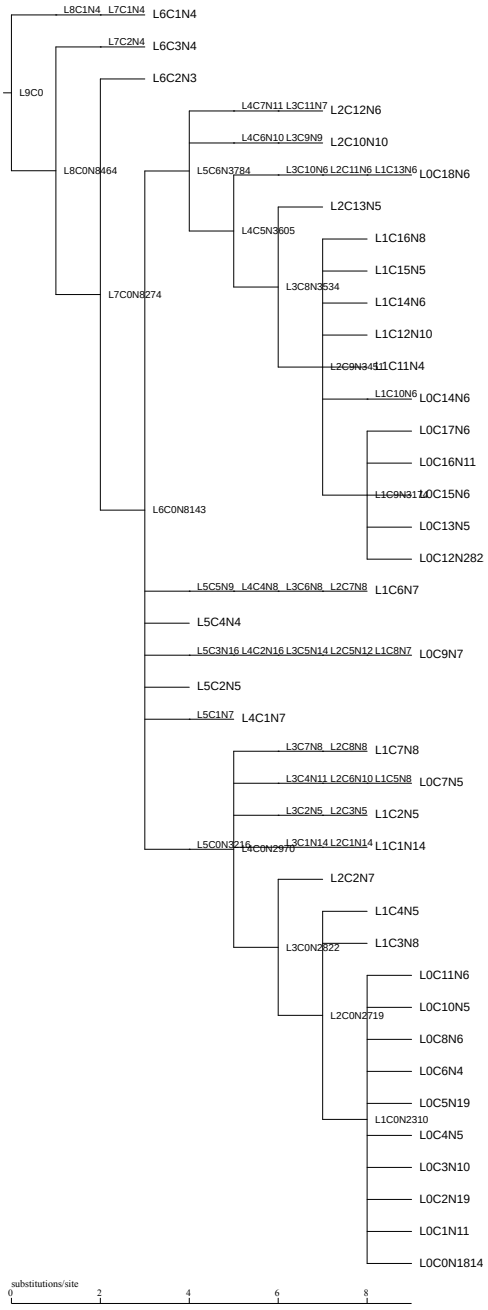


Figure 3.3 – A Newick tree with the cluster labels visible, each cluster is labeled by LXCYNZ where X is the layer number, Y the cluster number and Z the number of sequences in that cluster. The substitutions/site are automatically added by the Newick tools and have in our case no significance.

mutations and the other 20000, and hence both partitions diffused, as seen in the figures of the PCA 3.1 and as it can be seen in the figures of the tree 3.3.

Looking at figure 3.3 imagine we would like to identify the clusters *L5C0* containing 3216 sequences and *L5C6* containing 3784 clusters. Both are at the forefront of two large subtrees. In order to access the sequences in these clusters we have to convert layer 5 to FASTA files. Layer 5 exists by *MNHN-Tree-Tools* convention in a binary *split-set* file named with the suffix number *0005*, hence in our case, as demanded by the command in listing 3.8 in the folder *clusters* named *L0005*. In order to convert the sequences of layer 5 to *FASTA* files we use the tool *split_set_to_fasta* as described in section 4.13. We note here to the reader, that his personal tree, even using our commands might slightly differ due to different random number generator mechanisms on different systems, and that he might have to adapt cluster and layer numbers to the tree he has obtained. The *split_set_to_fasta* tool is called as follows:

```
mkdir fasta-layer-5
split_set_to_fasta test.fasta clusters/L0005 fasta-layer-5/C
```

Listing 3.11 – Converting a binary *split-set* to *FASTA* sequences

As such we have obtained the FASTA files for layer 5. The files for our clusters *L5C0* and *L5C6* are respectively stored in the files *fasta-layer-5/C-000* and *fasta-layer-5/C-006* in FASTA format. Using the same mechanism we can identify any sequence anywhere in the tree. Now one could open these files for instance with *SEAVIEW* and apply all kinds of statistical tools, or our *consens* tool highlighted in section 4.8 on this files.

But let us do something more interesting let us identify the two clusters in our PCA diagram, such as the one shown in figure 3.1. In order to do this we can use a tool called *split_set_to_projections*, further described in section 4.14. We run this tool likewise:

```
mkdir projections-layer-5
split_set_to_projections test.fasta test.pca 7 clusters/L0005 \
  projections-layer-5/p
```

Listing 3.12 – Converting a binary *split-set* to PCA projections

Analogous to the FASTA files we obtain the files containing the PCA projections of these clusters. Again in our case the projections for *L5C0* and *L5C6* reside in files *p-000* and *p-006* in folder *projections-layer-5*. We can right now use the *gnuplot* [5] software to locate them:

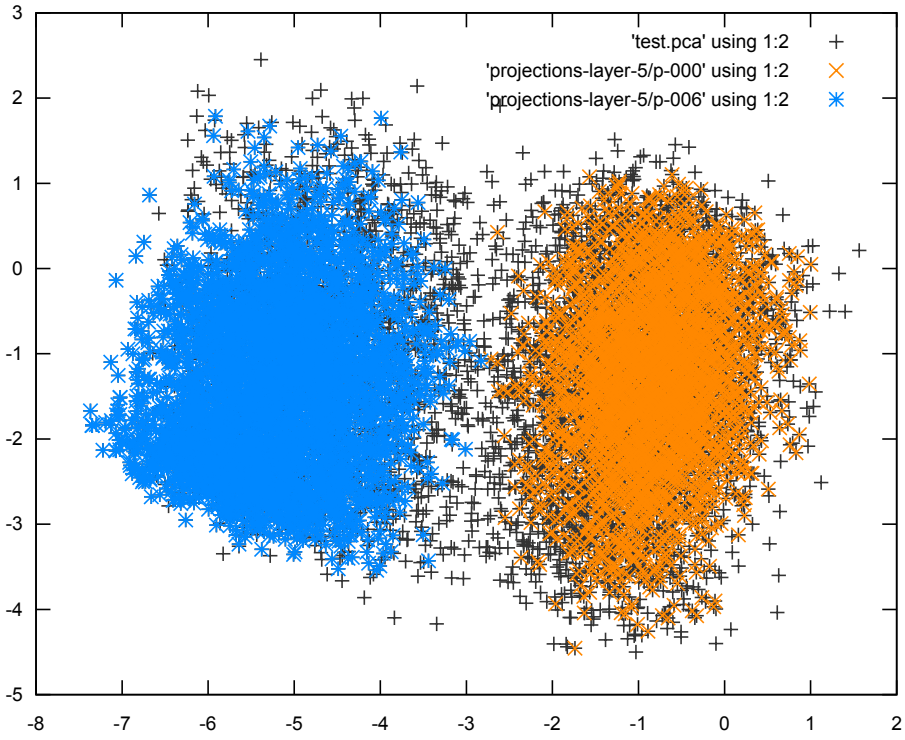


Figure 3.4 – The projections of cluster *L5C0* in orange, and the projections of cluster *L5C6* in blue projected onto the first two principal components

```
plot 'test.pca' using 1:2 lt rgb '#333333', \
'projections-layer-5/p-000' using 1:2 lt rgb '#FF8800', \
'projections-layer-5/p-006' using 1:2 lt rgb '#0088FF'
```

Listing 3.13 – Highlighting cluster projections with *gnuplot*

which generates in our case the figure 3.4. Note that as the algorithm is written in order to infer the tree structure and consensus (center of gravity) of clusters / sequence families and as such the clusters might appear small being projected in a PCA subspace. In such a case you can search for a more optimal cluster size changing the ϵ value around the layer in question. The ϵ values for each layer are found in the output file, in our case *cl-out* as requested in listing 3.8. You may find lines like the following:

```
Layer 5 has 7 clusters
  Coverage at layer 5: 0.704100
```

```
  Epsilon at layer 5: 1.100000
```

Listing 3.14 – Layer 5 description in our adaptive cluster run output file

as the algorithm is tree preserving. If you find that the clusters are to small for your application you can run a single *DBSCAN* run [2] with the tools *cluster_dbscan_X* tools shown in section 4.4, choosing an epsilon value just below the one of the following layer, which in this case is described to be:

```
Layer 6 has 4 clusters
  Coverage at layer 6: 0.815400
  Epsilon at layer 6: 1.180000
```

Listing 3.15 – Layer 6 description in our adaptive cluster run output file

This way more defuse sequences might be taken into account, just before the clusters merge due to the $\epsilon = 1.18$ density criteria. You can therefore oversample the tree.

Let us further investigate the structure of the tree, and let us say we would like to see how a single sequence flows through, and where it appears in the tree. In order to achieve such a task we have the *three_map_for_sequence* tool as outlined in section 4.22. Imagine that we are interested in finding the following sequence that resides in our initial dataset in our tree:

```
>sequence_6773
CGGCTTACGAACTGCCGCTTGTTTCACCTGGGCACAGTGTACCACGCAAA
AGATGGAGTTACTCGCGGACAAATAGGGAGCGTTGTGTCCGGTGAGTTAG
```

Listing 3.16 – A sequence of interest

We hence save this sequence into a separated FASTA file called *target.fasta*. We can now use the the *tree_map_for_sequence* tool to create a map for the Newick tools by issuing the following command:

```
tree_map_for_sequence target.fasta test.fasta '#FF8800' clusters/L* > target.map
```

Listing 3.17 – Mapping the *target.fasta* sequence in the Newick tree

This allows us to color the clusters in the tree that contains the target sequence. To generate a SVG file you can use the Newick tools command:

```
nw_display -s -b 'opacity:0' -w 600 tree.dnd -c target.map > tree-seq6773.svg
```

Listing 3.18 – Drawing the dendogram highlighting the sequence in *target.fasta*

which results in a dendogram as shown in figure 3.5.

3.5 Comparison to Ground Truth

In this section we show you how you can investigate not just the location of one single sequences but of a whole ensemble of sequences within the tree obtained by an adaptive clustering run. If you know about certain sequences that form a group in your own dataset you can follow us on this route, generating your own clusters that you might want to investigate in your own trees. The dataset generated with the command highlighted in listing 3.1 is partitioned as we could also suggest from the tree into two subgroups. The *virtual_evolution* (c.f. section 4.24) tool has generated two partitions of 5000 sequences. As the partitions are written out sequential by the *virtual_evolution* tool the first 5000 sequences belong to the first partition, the next 5000 sequences belong to the second partition. We can thus generate an artificial clustering or binary *split-set* in order to further investigate the two known partitions with the *MNHN-Tree-Tools*. In order to generate such a *split-set* we will use the *split_set_from_annotation* tool outlined in section 4.17. In order to do this we first have to generate the annotation file. Using the *bash* [9] shell on a Unix/Linux system we can perfrom this in the following way:

```
for ((i=0;i<5000;i++)); do echo "one" >> gt-annotation; done
for ((i=0;i<5000;i++)); do echo "two" >> gt-annotation; done
echo "one" >> unique-gt-annotation
echo "two" >> unique-gt-annotation
```

Listing 3.19 – Generating an annotation file for *split_set_from_annotation*

which creates us the *gt-annotation* file that contains 5000 lines with just the single word *one* on each line followed by an other 5000 lines with the single word *two* on each line. We further create the *unique-gt-annotation* file containing just two lines with the statements *one* and *two*. The *unique-gt-annotation* file is used, first to describe that statements that indicate the clusters to be formed, in our case *one* and *two* and second to order the clusters in the resulting *split-set* file. As *one* fills the first line, and *two* the seconds in the *unique-gt-annotation* file we know that sequences marked as *one* in the *gt-annotation* file will be

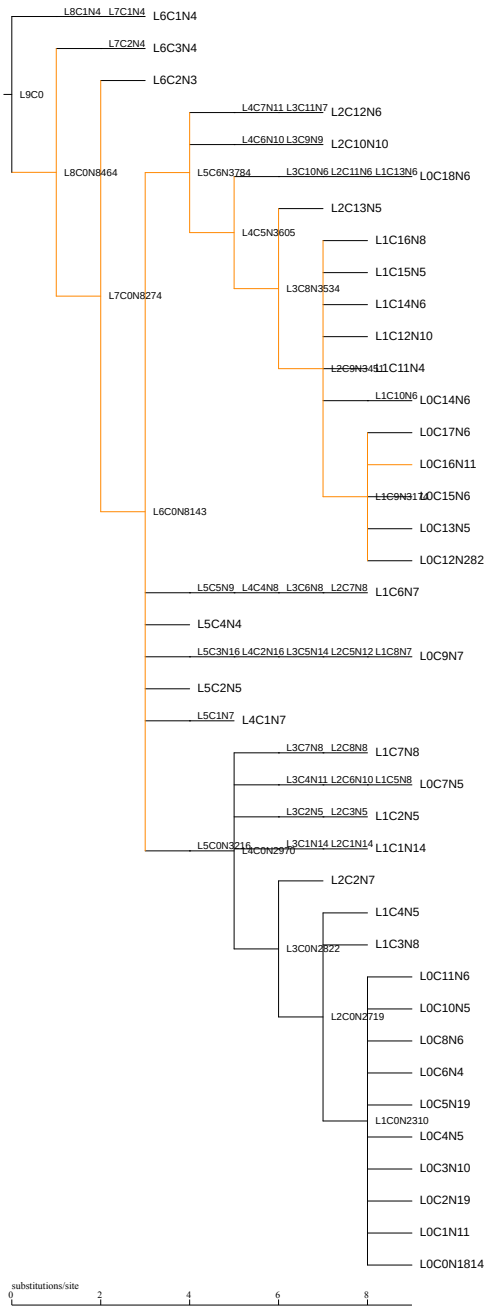


Figure 3.5 – The tree, tracing down a target sequence highlighted by the *tree_map_for_sequence* tool.

part of the first clusters while those marked to be *two* will be part of the second cluster. With these files at hand we can execute the *split_set_from_annotation* command:

```
split_set_from_annotation test.fasta gt-annotation unique-gt-annotation \
  ground-truth
```

Listing 3.20 – Creation of a *split-set* cluster file by manual annotation

which generates us the ground truth binary *split-set* indicating the two clusters.

Knowing that the first 5000 sequences form a partition and the second 5000 sequences form an other partition as created by using our *virtual_evolution* tools we would like to know where these partitions are to be found in the tree. In order to investigate this matter we use the *tree_map_for_split_set* tool as highlighted in section 4.23:

```
tree_map_for_split_set ground-truth test.fasta 1 2 '1,#000000' \
  clusters/L* > tree-one.map
tree_map_for_split_set ground-truth test.fasta 1 2 '2,#000000' \
  clusters/L* > tree-two.map
```

Listing 3.21 – Creating a Newick tools [8] map file for a *split-set*

which yields us the *tree-maps* that we can use with the Newick tools [8] to outline the different datasets in the trees. In order to do this we generate two SVG files for each tree executing:

```
nw_display -s -b 'opacity:0' -w 600 tree.dnd -c tree-one.map > tree-one.svg
nw_display -s -b 'opacity:0' -w 600 tree.dnd -c tree-two.map > tree-two.svg
```

Listing 3.22 – Using the Newick tools [8] to visualize partitions

together with the dendogram created in listing 3.9. The two resulting trees are shown in figure 3.6.

Finally we can quantize if our tree correctly maps the two clusters by applying our *Pureness* calculation. This allows us to check weather we obtain the same clusters as the ground truth and weather clusters are a mixture of the ground truth or not. For the details of the Pureness, and other indices to evaluate the quality of a tree against ground truth the reader is referred to section 4.32. In order to perform this quality measurement we execture the *tree_pureness* tool:

```
tree_pureness test.fasta ground-truth clusters/L* > pureness
```

Listing 3.23 – Evaluating a tree against ground truth with the *tree_pureness* tool

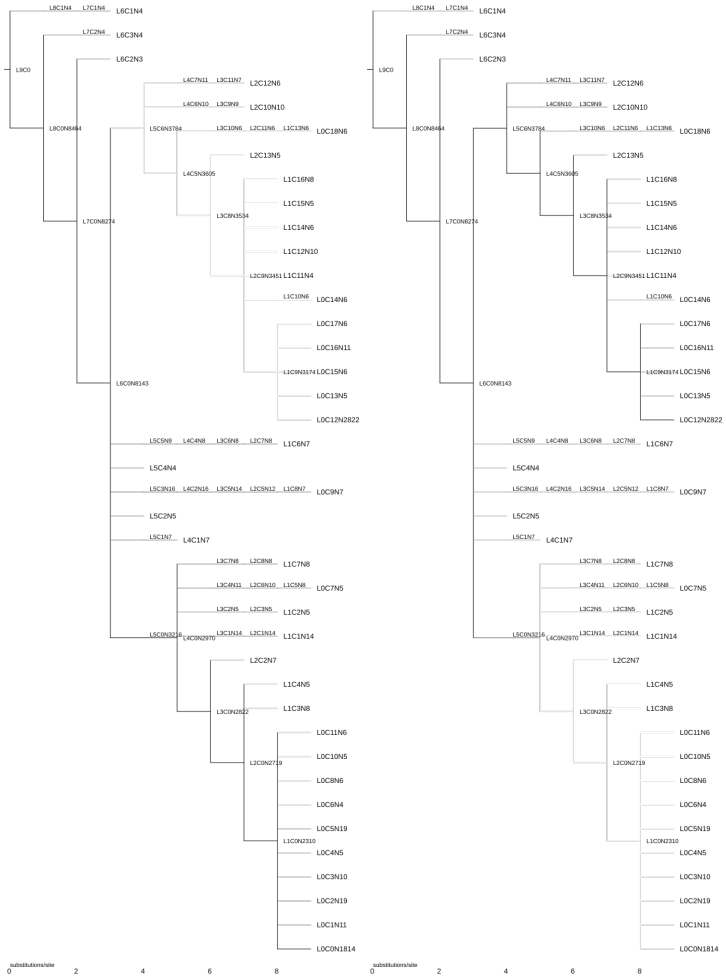


Figure 3.6 – Tracing down clusters using the *tree_map_for_split_set* tool. On the left we have the first partition while on the right the second partition is shown. Using the logarithmic representation we can also easily track down the core clusters *L0C0* and *L0C12* that have generated these two families, forming these two subtrees (lower on the right, upper on the left).

from where we obtain the following table:

```
0.001103        8.500000        0.052632
0.250433        7.500000        0.117647
0.134946        6.000000        0.214286
0.002904        5.000000        0.166667
0.002793        3.000000        0.250000
0.002923        2.500000        0.285714
0.978448        1.000000        0.500000
0.981508        0.500000        0.666667
0.969282        0.000000        0.500000
0.973321        0.500000        1.000000
```

Listing 3.24 – The output of the *Pureness* table

where the first column is the Purness as in equation 4.23, the second the Cluster Correspondence according to equation 4.24 and finally the Impure over Pures as outlined in equation 4.25. The table shows the values for each layer from the leaf nodes on top to the root of the three on the bottom. We see that in Layer 5 (we count from the top, and note that the first line corresponds to Layer 0) we have a very good Pureness of 0.003 which counts only for the already impure clusters, and that only about a quarter 28% of the clusters (as shown in the third column) are impure. The middle column shows that at this moment the tree, nevertheless is not only composed of two Datasets, but contains more clusters in this Layer. A quick look at the Tree in figure 3.3 shows that these clusters have very few elements and have formed due to the fluctuations caused by random diffusion. They would have been filtered away by a more aggressive *minPoints* value in the adaptive clustering run.

4 | Tools

In this chapter we describe each tool in detail, how to use it, the algorithm used to implement it, and algorithmic specificities.

4.1 fasta2kmer

4.1.1 General

The *fasta2kmer* tool implements the calculation of k-mer frequencies for each sequence to be found in a FASTA [1] file. The tool can calculate both k-mer frequencies for nucleic and protein sequences. k-mers are all possible sequences for k letters. The frequencies tell how often each of these k-mers is found in the entire sequence.

4.1.2 Usage

```
fasta2kmer [fasta-file] [kmer-length] [number-of-threads] [protein?] > kmerbase
```

Listing 4.1 – Calling the *fasta2kmer* tool

The *fasta2kmer* tool has the following arguments:

1. *fasta-file* The FASTA file containing the sequences to calculate k-mer frequencies of.
2. *kmer-length* The length of the k-mers to calculate frequencies from.
3. *number-of-threads* The number of individual threads to be available for this computation. A good choice is the number of logical cores available on the machine in question.
4. *protein?* Set 0 for nucleic sequences, 1 for protein sequences.

5. *kmerbase* The output file to write kmer frequencies for each sequence to. The format of this file is tabulator separated. The first entry in each line is *sequence_N* where N is the sequence number attributed counting from 0 to N by occurrence of sequences in the FASTA file. The following entries the line, purely numeric are the frequencies of the k-mer in questions.

4.1.3 Example

```
fasta2kmer test.fasta 5 8 0 > test.kmer
```

Listing 4.2 – Example call *fasta2kmer* tool

In this case one calculates 5-mer frequencies for the file test.fasta using 8 threads. test.fasta contains nucleic sequences. The output ist stored in test.kmer.

4.1.4 Algorithm

The nucleic k-mers are calculated using a binary scheme. As the nucleic letters contain only four members A, C, G, T they can be represented by 00, 01, 10, 11 in binary form. To generate all possible k-mers we generate a binary number of $2k$ in length containing only ones. I.e. for $k = 3$ we generate $111111 = 63$ and count down to 0 generating all 64 3-mers by comparing the binary representation with the letter code.

The protein code uses a different scheme generate proteic k-mers. A recursive function called k times shifting position counting to from 0 to 19 is called generating all possible protein k-mers for the length k.

The frequencies are calculated by comparing the letters at each overlay position of the sequence to calculate frequencies for. An early stop condition for a mismatching letter is implemented. The comparison is performed in multithreaded fashion distributing the number sequences to calcualte k-mer frequencies from across the number of threads requested.

4.1.5 Implementation

The algorithms for the tool are implemented in *kmers.c* The interface to the algorithms is implemented in *fasta2kmer.c*

4.2 kmer2pca

4.2.1 General

The *kmer2pca* tool implements the calculation of and projection onto principal components. The tools yields the projections onto the principal components as well as the eigenvalues of covariance matrix.

4.2.2 usage

```
kmer2pca [kmer-file] [projections] [eigenvalues] [dimensions] [n-threads]
```

Listing 4.3 – Calling the *kmer2pca* tool

The *kmer2pca* tool has the following arguments.

1. *kmer-file* A file containing k-mer frequencies as calculated by the fasta2kmer tool presented in section 4.1
2. *projections* The file projections onto the principal components are written to. This file contains tabulator separated values, each line corresponds to the line in k-mer frequencies input file.
3. *eigenvalues* The file the eigenvalues of covariance matrix are written to. The eigenvalues are sorted from smallest to largest.
4. *dimensions* The number of principal components to project on and as such the number of dimensions of the resulting subspace. The k-mers are projected onto the [dimension] principal components corresponding to [dimension] highest eigenvalues.
5. *n-threads* The number of threads to be made available for this computation. A good choice is the number of logical cores available on the machine.

4.2.3 example

```
fasta2kmer test.kmer test.pca test.ev 7 8
```

Listing 4.4 – Example call *kmer2pca* tool

In this example we calculate the projections into a seven dimensional subspace spun by the 7 principal components with the highest variance. The projections onto these 7 principal components of the k-mers stored in test.kmer are written to test.pca. The eigenvalues of the covariance matrix built from the kmers in test.kmer are written to the test.ev file. 8 threads are used to build the covariance matrix.

4.2.4 algorithm

The PCA is straight forward implemented in its most common form. Having the vectors of k-mer frequencies for each sequence at hand we can call each k-mer a feature, of which we have the number of sequences samples. In order perform PCA we first need to obtain a feature covariance matrix, and hence calculate the covariance between each pair of k-mers in the dataset. We first calculate the mean of all k-mers, let us call a single k-mer X:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i, \tag{4.1}$$

where n is the number of sequences. For two different k-mers, let us call them X and Y we can than calculate the covariance:

$$\sigma_X \sigma_Y = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y}). \tag{4.2}$$

and further built the covariance matrix for all available covariances from all available k-mers.

$$C = \begin{bmatrix} \sigma_1\sigma_1 & \cdots & \sigma_1\sigma_n \\ \vdots & \ddots & \vdots \\ \sigma_n\sigma_1 & \cdots & \sigma_n\sigma_n \end{bmatrix}. \tag{4.3}$$

From this covariance matrix solve the eigenvalue equation:

$$Cv_i = \lambda_i v_i, \tag{4.4}$$

where v_i are the eigenvectors or in the sense of PCA the principal components while λ_i are the associated eigenvalues.

Once the principal components are obtained they are sorted together with the associated eigenvalues. In order to calculate the projections a sample S and hence a sequence in its k-mer representation has to be projected onto the principal components v_i

with the highest eigenvalues. The number of projections is given by the *dimension* parameter of the tool. The projections p_i are the simple inner product of a k-mer representation of a sequence S with the eigenvector v_i,

$$p_i =< v_i|S > . \tag{4.5}$$

As such we have effectively diminished the dimensionalty to the number of *dimensions* as selected.

The algorithm is implemented by parallization on different levels. The algorithm spends for a great number of samples most of its time calculating the matrix elements of equation (4.2). The matrix itself rests in most cases of a small rank, the number of individual k-mers. As such solving the eigenvalues and eigenvectors (principal components) in equation (4.4) is especially for k-mers of small k fast. Hence, we focused our efforts on the calculation of the matrix elements in (4.2). As the covariance matrix is symmetric we calculate only the half of the elements transposing them into the other half. Each row of the matrix is calculated in its own thread, allowing us to calculate several rows on several cores in parallel. Besides this we used Single Instruction Multiple Data (SIMD) instructions in evaluating equation (4.2). We do this in calculating partial sums:

$$\sigma_X\sigma_Y[1] = \sum_{i=1}^{n/4}(X_{[(i+4)n/4]} - \bar{X})(Y_{[(i+4)n/4]} - \bar{Y}) \tag{4.6}$$

$$\sigma_X\sigma_Y[2] = \sum_{i=2}^{n/4}(X_{[(i+4)n/4]} - \bar{X})(Y_{[(i+4)n/4]} - \bar{Y}) \tag{4.7}$$

$$\sigma_X\sigma_Y[3] = \sum_{i=3}^{n/4}(X_{[(i+4)n/4]} - \bar{X})(Y_{[(i+4)n/4]} - \bar{Y}) \tag{4.8}$$

$$\sigma_X\sigma_Y[4] = \sum_{i=4}^{n/4}(X_{[(i+4)n/4]} - \bar{X})(Y_{[(i+4)n/4]} - \bar{Y}) \tag{4.9}$$

$$\sigma_X\sigma_Y = \sum_{i=1}^{4}\sigma_X\sigma_Y[i]. \tag{4.10}$$

Further the partial sums are implemented by the Kahan summation algorithm [10], avoiding floating point errors in summing a large number of values.

Calculating the eigenvectors and eigenvalues is performed by calling the optimized *dsyev* routine from the LAPACK libraries [11].

4.2.5 Implementation

Both the algorithm and the interface are implemented in *kmer2pca.c*.

4.3 pca_visual_extract

4.3.1 General

The PCA visual extract tool allows you to visualize a sequence dataset, that has been transformed to a k-mer representation that has been projected onto its principal components. Its core functionality is a visual selection mechanism. In order to obtain the necessary sequence representations the reader is referred to the *fasta2kmer* and *kmer2pca* tools highlighted in sections 4.1 and 4.2 respectively.

4.3.2 Usage

The Interface for the *pca_visual_extract* tool can be brought up by calling:

```
pca_visual_extract [fasta] [pca] [dimensions] [first-dim] [second-dim] \
  [out-fasta]
```

Listing 4.5 – Calling the *pca_visual_extract* tool

with the arguments being:

1. *fasta* A FASTA file that has been processed by the *fasta2kmer* and *kmer2pca* tools (c.f. sections 4.1 and 4.2) to create a representation of the dataset in a PCA subspace.
2. *pca* The projections onto the principal components as generated by *kmer2pca*.
3. *dimensions* The number of principal components the sequences have been projected onto. The number of dimensions of the PCA subspace.
4. *first-dim* The first principal component to visualize.

5. *second-dim* The second principal component to visualize.
6. *out-fasta* A FASTA file where the selected sequences will be written to.

Here *first-dim* and *second-dim* allow you to choose an arbitrary plane, spun by two principal components to select your sequences from in a visual manner. Once the call to the program was issued on the shell and all the arguments are coherent, a graphical user interface as shown in figure 4.1 is presented on the screen. On the top of figure 4.1 you see a window in which you can draw a rectangle using your mouse cursor. The terminal as shown below than shows you how many sequences you have selected. Once you are fine with your selection you might hit the Enter key in order to write out your selection to a FASTA file.

4.3.3 Example

```
pca_visual_extract test.fast test.pca 7 2 3 /tmp/out.fasta
```

Listing 4.6 – Example of the *pca_visual_extract* tool

Will bring up an interface such as the one shown in figure 4.1, representing the sequences in *test.fasta* in their 7 dimensional PCA subspace as stored in *test.pca*. A two dimensional image representing the second and third principal component is drawn. Using the mouse one can draw a rectangle selecting sequences. Once the right sequences are selected hitting the Enter key will store them to */tmp/out.fasta*.

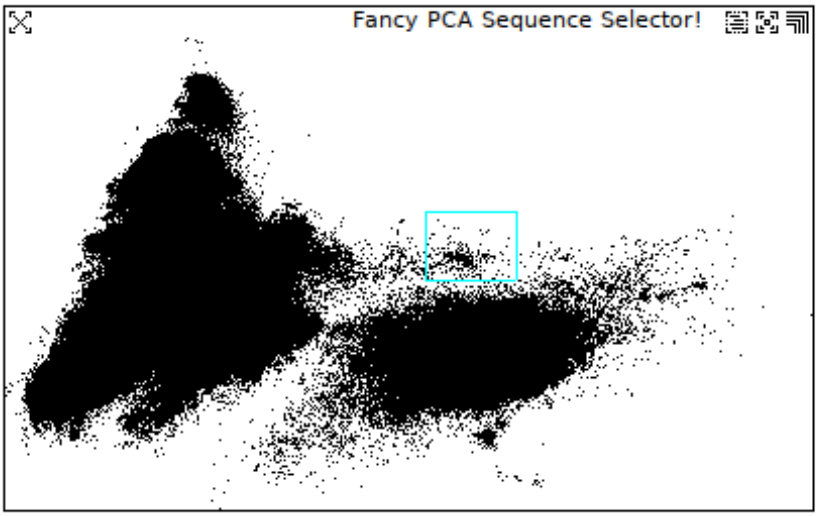
4.3.4 Implementation

The tool is implemented in *pca_visual_extract.c* and makes use of the Simple Direct Layer 2 (SDL2) library for its drawing routines and portability across systems.

4.4 cluster_dbscan_X

4.4.1 General

The *cluster_dbscan_PCA*, *cluster_dbscan_kmerL1*, *cluster_dbscan_kmerL2*, *cluster_dbscan_SW* and *cluster_dbscan_SW_GPU* implement clustering of a dataset of sequences using the DBSCAN [2] algorithm.




```
haschka@epr:/mnt/data/data/uni/mnhn/mnhn-tools-manual
F50_tree_from_annotation.pdf
F50_tree_from_annotation.svg
SILVA_138_SSURef_NR99_tax_silva_trunc.fasta
SILVA_138_SSURef_tax_silva_trunc-dna-F200.split_set_from_swarm
SILVA_138_SSURef_tax_silva_trunc-dna-for-swarm.fasta
SILVA_138_SSURef_tax_silva_trunc-dna-for-swarm.swarm
SILVA_138_SSURef_tax_silva_trunc-dna-for-swarm.uniq
SILVA_138_SSURef_tax_silva_trunc-dna.fasta
SILVA_138_SSURef_tax_silva_trunc-dna.pca-7v
SILVA_138_SSURef_tax_silva_trunc-dna.split_set_from_swarm
SILVA_138_SSURef_tax_silva_trunc-dna.uniq
SILVA_138_SSURef_tax_silva_trunc.5mer
SILVA_138_SSURef_tax_silva_trunc.fasta
SILVA_138_SSURef_tax_silva_trunc.uniq
full_group_label
full_group_label_uniq
full_group_split_set
haschka@epr /mnt/data/data/uni/mnhn/mnhn-tools-manual $ ../code/bin/pca_visual_e
xtract ../silva/SILVA_138_SSURef_tax_silva_trunc-dna.fasta ../silva/SILVA_138_SS
URef_tax_silva_trunc-dna.pca-7v 7 1 2 /tmp/out.fasta
Calculating...
Selected 510 Sequences !
Hit enter to save to fasta file: /tmp/out.fasta
```

Figure 4.1 – The color inverted interface of the *pca_visual_sequence_selector*.

The different variants implement DBSCAN clustering using different sequence-sequence distances and computational platforms.

4.4.2 Usage

```
cluster_dbscan_pca [fasta] [pcaproj] [dimensions] [epsilon] [minPoints] \
   [outfiles-fasta-prefix] [outfiles-values-prefix]

cluster_dbscan_kmerL1 [fasta] [kmers] [epsilon] [minPoints] \
   [outfiles-fasta-prefix]

cluster_dbscan_kmerL2 [fasta] [kmers] [epsilon] [minPoints] \
   [outfiles-fasta-prefix]

cluster_dbscan_SW [fasta] [epsilon] [minPoints]
   [outfiles-fasta-prefix]

cluster_dbscan_SW_GPU [fasta] [epsilon] [minPoints] \
   [outfiles-fasta-prefix]
```

Listing 4.7 – Calling the *cluster_dbscan* tools

Where the different tools perform as disguised by their name:

1. *cluster_dbscan_pca*: Clustering of sequences in a PCA subspace such as the one obtained using the *kmer2pca* tool described in section 4.2.
2. *cluster_dbscan_kmerL1*: Clustering of sequences in a k-mer representation using the L_1-norm, also known as the Manhattan distance, to calculate the distance between k-mer frequency vectors representing sequences.
3. *cluster_dbscan_kmerL2*: Clustering of sequences in a k-mer representation applying the L_2-norm, also known as the Euclidean distance, to calculate the distance between k-mer frequency vectors representing sequences.
4. *cluster_dbscan_SW*: Clustering of sequences using the Smith-Waterman distance measure.
5. *cluster_dbscan_SW_GPU*: Clustering of sequences using the Smith-Waterman distance measure on an OpenCL supported device, for instance a GPU.

The tools have the following arguments:

1. *fasta*: A FASTA file containing the sequences to be clustered. In the case of Smith-Waterman clustering the FASTA file has to consist of nucleic sequences as clustering of amino acid sequences under a Smith-Waterman distance

is not implemented. In case the Smith-Waterman distance is evaluated on GPUs the maximum sequence length is currently limited to 300 nucleotide bases.

2. *pcaproj*: The projections onto the principal components as calculated by the *kmer2pca* tool.
3. *dimensions*: The number of dimensions, principal components found in the file.
4. *epsilon*: The epsilon neighborhood radius as used by the DBSCAN [2] algorithm.
5. *minPoints*: The minimum number of sequences to be found within an epsilon neighborhood in order to form or expand a cluster according to the DBSCAN algorithm.
6. *outfiles-fasta-prefix*: The DBSCAN based tools all generate a FASTA file per cluster found. With this argument you can tell the program where to store those clusters. The number of clusters to be generated is capped at 500 clusters. The resulting files will be named according to your prefix with a number at the end for each different cluster.
7. *outfiles-values-prefix*: If this parameter is added to the PCA version of the DBSCAN implementation the clusters are not only stored as individual FASTA files but also as files containing the projections onto the principal components for the clustering in question. This allows for visual representations of the PCA projections.

4.4.3 Algorithm

The utility implements the DBSCAN algorithm in a straight forward fashion as described in its original article [2]. The DBSCAN algorithm is perfect in finding *density connected* subsets of a dataset. The algorithm as such finds all connected ensembles in a dataset that exceed a certain density:

$$\rho = \frac{n}{V(\epsilon)} \tag{4.11}$$

where V is the Volume within an epsilon neighborhood defined by the radius ϵ. n corresponds to the number of sequences within the aforementioned volume, and is defined using the *minPoints* parameter. The formula for the volume depends on the metric

used, but is hypercubic for L_1 measurements, and hyperspheric for L_2 measurements. Sequences scattered in sequence space with a density below ρ are found to be outliers and will not be taken into account and are hence not to be found in the resulting clusters.

The different distance measures in the *region expand method* as described in the original paper use various means of optimization and parallization. In the case of PCA, kmerL1, kmerL2 distance methods up to eight distances between sequences are calculated in parallel making use of the AVX SIMD instructions. In the case of the Smith Waterman distance measure an OpenCL (GPU) method is available allowing to efficiently calculate multiple distances in parallel on OpenCL devices such as GPUs.

The Smith Waterman algorithm calculates the Smith Waterman matrix and defines the maximum value to be found within the matrix to be the distance between two sequences. So far no implementation a Smith Waterman algorithm using amino acid sequences exists in *MNHN-Tree-Tools*. The algorithm builds the matrix using the recurrence relation:

$$\begin{aligned} s_1 &= M(j-1, i-1) + F(A(i), B(j)), \\ s_2 &= M(j, i-1) - G, \\ s_3 &= M(j-1, i) - G, \\ M(i,j) &= \max(s_1, s_2, s_3). \end{aligned} \tag{4.12}$$

where A and B are two sequences of the dataset to be compared. $A(i)$ is the i-th letter of sequence A and $B(j)$ is the j-th letter of sequence B.

$$F(A(i), B(j)) = \begin{cases} A(i) = B(j) : 4 \\ A(i) \neq B(j) : -4, \end{cases} \tag{4.13}$$

is the comparison relation. G is the gap penalty defined to be 3 and $M(i,j)$ the Sith Waterman matrix yielding the distance:

$$d(A, B) = \max(M(i,j)); \tag{4.14}$$

In the case of the GPU implementation only three subsequent rows of the matrix are stored in memory evaluating the maximum on the fly calculating row after row of the Smith Waterman Matrix. This shall allow to use more local memory that is closer to the compute unit, if not only local registers on the graphics card for the computation.

4.4.4 Example

```
cluster_dbscan_PCA test.fasta test.pca 7 0.02 4 /tmp/outf /tmp/outv
```

Listing 4.8 – Example of the *cluster_dbscan_PCA* tool

This example performs DBSCAN based clustering using L_2 distances in a 7 dimensional subspace with subspace projections stored in the test.pca file. The input values for the DBSCAN run are $\epsilon = 0.02$ and minpoints $n = 4$. Each resulting cluster will be stored in /tmp/ in FASTA format with a file name of outf$N$, and as projections onto the principal components outv$N$, where N is an individual number for each cluster.

4.4.5 Implementation

The DBSCAN algorithm for all variants is implemented in *dbscan.c* The Interface for the Smith-Waterman variants is implemented in *cluster_dbscan_SW.c*. The Interface for k-mer distance based L_1 and L_2 variants is implemented in *cluster_dbscan_kmer.c*. The Interface for the PCA variant is implemented in *cluster_dbscan_pca.c*.

4.5 compareSW

4.5.1 General

The *compareSW* tool calculates the mean distance and standard deviation between two datasets using the Smith Waterman distance. This tool only works with nucleotide sequences.

4.5.2 Usage

One can call the *compareSW* tool like:

```
compareSW [fasta1] [fasta2] [n-threads]
```

Listing 4.9 – Calling the *compareSW* tool

with the following arguments:

1. *fasta1*: A FASTA file containing a set of sequences.
2. *fasta2*: An other FASTA file containing a set of sequences.
3. *n-threads*: The number of threads that this computation might use.

4.5.3 Algorithm

The tool calculates the mean and the standard deviation of the distances between the sequences in the two datasets. If the first dataset has n sequences and the second j, the number of total distances calculated is hence, $N = nj$. The Smith Waterman algorithm is the same as outlined in section 4.4.3.

4.5.4 Example

```
compareSW test1.fasta test2.fasta 8
```

Listing 4.10 – Example of the *compareSW* tool

This example computes the mean distance between the sequences in test1.fasta and test2.fasta using 8 cores.

4.5.5 Implementation

The functions to calculate the mean and standard deviation are implemented in *comparison.c*. The Interface is implemented in *compareSW.c*.

4.6 find_satellite

4.6.1 General

The *find_satellite* tool is a filtering tool narrowing down a FASTA dataset of similar sequences. The filter can filter sequences by length, and by Smith Waterman distance to a template sequence. The Smith Waterman algorithm is implemented as shown in 4.4.3. This tool only works with nucleotide sequences.

4.6.2 Usage

The tool is to be called like:

```
find_satellite [fasta-ds] [fasta-target] [size] [size+] [size-] \
  [sw-dist] [n-threads] > [fasta-filtered]
```

Listing 4.11 – Calling the *find_satellite* tool

where the arguments are:

1. *fasta-ds* A FASTA dataset of the sequences that should be searched for according to the pattern selected by the following arguments.
2. *fasta-target* A FASTA file holding the target sequence from which the Smith Waterman distance will be calculated.
3. *size* The length of the sequence to be searched for with the tolerance intervals given be the subsequent arguments.
4. *size+* How many nucleotides above the size sequences are accepted.
5. *size-* How many nucleotides below the size sequences are accepted.
6. *sw-dist* The maximum Smith Waterman distance from the target sequence to accept sequences.
7. *n-threads* The number of parallel threads this computation may use.
8. *fasta-filtered* A FASTA file where the sequences that met the filtering criteria above are written to.

4.6.3 Example

```
find_satellite test.fasta target.fasta 100 5 3 20 8 > filtered.fasta
```

Listing 4.12 – Example of the *find_satellite* tool

Here the tool searches for sequences in *test.fasta* that are less then a Smith Waterman distance of 20 away from the sequence stored in *target.fasta*. Sequences that are between 97 and 105 nucleotides long are accepted. The resulting sequences are stored to *filtered.fasta*

4.6.4 Implementation

The tools interface is implemented in *find_satellite.c*. The mechanism is implemented in *filter.c*.

4.7 silhouette

4.7.1 General

The *silhouette* tool calculates the silhouette index comparing two clusters or datasets stored in two FASTA files. The *silhouette* tool only works with nucleotide sequences.

4.7.2 Usage

One can call the *silhouette* tool like:

```
silhouette [fasta1] [fasta2] [n-threads]
```

Listing 4.13 – Calling the *silhouette* tool

with the following arguments:

1. *fasta1*: A FASTA file containing a set of sequences.
2. *fasta2*: An other FASTA file containing a set of sequences.
3. *n-threads*: The number of threads that this computation might use.

4.7.3 Algorithm

The tool calculates the silhouette index comparing the distances internal to each dataset (distances between sequences within a FASTA file), with the distances external to each dataset (distances between sequences between two FASTA files).

The silhouette index is calculated using the following formula:

$$S = \frac{1}{n_1} \sum_{i=1}^{n_1} s_i, \tag{4.15}$$

with s_i defined to be:

$$s_i = \begin{cases} a_i < b_i : 1 - a_i/b_i \\ a_i = b_i : 0 \\ a_i > b_i : b_i/a_i - 1 \end{cases} \tag{4.16}$$

and a_i being the internal measure for distances to the i-th sequence A_i found in the first FASTA file *fasta1* and hence,

$$a_i = \frac{1}{n_a} \sum_{j=1}^{n_A} d_{\mathrm{SW}}(A_i, A_j). \tag{4.17}$$

Likewise b_i is the external measure for distances to the i-th sequence A_i in the first FASTA file and sequences B found in the second FASTA file. We write as such:

$$b_i = \frac{1}{n_b} \sum_{j=1}^{n_B} d_{\text{SW}}(A_i, B_j), \tag{4.18}$$

with $d_{SW}(X, Y)$ being the Smith Waterman distance between two sequences X and Y as defined in section 4.4.3. n_A and n_B are the number of sequences found in the first A, and second B FASTA file.

4.7.4 Example

```
silhouette test1.fasta test2.fasta 8
```

Listing 4.14 – Example of the *silhouette* tool

This example calculates the silhouette index between the dataset of sequences found in *test1.fasta* and the dataset found in *test2.fasta*.

4.7.5 Implementation

The functions to calculate the silhouette index are implemented in *comparison.c*. The interface is implemented in *silhouette.c*.

4.8 consens

4.8.1 General

The *consens* tool calculates a consens sequence from sequences found in a fasta file. Further the *consens* tool provides detailed information about the distribution of nucleotide sequences in a datasets. The sequences have to be aligned or quasi-aligned before calling the *consens* tool. The tool currently only works with nucleotide sequences.

4.8.2 Usage

The *consens* tool can be called like:

```
consens [fasta]
```

Listing 4.15 – Calling the *consens* tool

where the *fasta* argument is a FASTA file to perform the *consens* calculation on.

4.8.3 Example

```
consens test.fasta
```

Listing 4.16 – Example of the *consens* tool

calculates the consens of the sequences found in test.fasta and presents an output like the following:

```
Base       A          C          G          T       A       C       G       T
G         10          5        230          5 0.04000 0.02000 0.92000 0.02000
C         18        213          8         11 0.07200 0.85200 0.03200 0.04400
T          0          0          0        250 0.00000 0.00000 0.00000 1.00000
T          0          4          0        246 0.00000 0.01600 0.00000 0.98400
C          0        246          0          4 0.00000 0.98400 0.00000 0.01600
T          2          4          1        243 0.00800 0.01600 0.00400 0.97200
T          3          2          6        239 0.01200 0.00800 0.02400 0.95600
G          5          0        240          5 0.02000 0.00000 0.96000 0.02000
A        245          3          0          2 0.98000 0.01200 0.00000 0.00800
A        237          2         10          1 0.94800 0.00800 0.04000 0.00400
G          8          0        241          1 0.03200 0.00000 0.96400 0.00400
G          3          0        241          6 0.01200 0.00000 0.96400 0.02400
G          9          4        234          3 0.03600 0.01600 0.93600 0.01200
A        244          2          2          2 0.97600 0.00800 0.00800 0.00800
A        246          3          0          1 0.98400 0.01200 0.00000 0.00400
A        240          1          7          2 0.96000 0.00400 0.02800 0.00800
G         15         11        222          2 0.06000 0.04400 0.88800 0.00800
```

Listing 4.17 – Example output of the *consens* tool

where you are presented in the first column with the consensus sequence, from the second to the fifth column with the absolute number of the according bases found and from the sixth to the ninth column with the percentage of each bases occurring at this position. The final line also shows some statistics:

```
MEAN MAX ACCURACY: 0.897734 SIGMA: 0.091085
```

Listing 4.18 – Example statistics output of the *consens* tool

where *MEAN MAX ACCURACY* is the mean of the highest percentage between the four nucleotide bases for each position and *SIGMA* the square of the standard deviation.

4.9 sequence_multiplicity

4.9.1 General

The *sequence_multiplicity* finds unique sequences in a FASTA file and allows one to know how often a unique sequence appears

in the FASTA file.

4.9.2 Usage

One can call the *sequence_multiplicity* tool like:

```
sequence_multiplicity [fasta] > [fasta-out]
```

Listing 4.19 – Calling the *sequence_multiplicity* tool

with the arguments:

1. *fasta* a FASTA file to find unique sequences in.
2. *fasta-out* the unique sequences in FASTA format.

4.9.3 Example

```
sequence_multiplicity test.fasta > out.fasta
```

Listing 4.20 – Example of the *sequence_multiplicity* tool

Finds unique sequences in *test.fasta* and writes them into *out.fasta*. Each sequence in the *out.fasta* file contains a suffix number telling you how many times this sequence occurred in the original dataset.

```
>sequence_2_2344
```

Listing 4.21 – Example a line in *out.fasta*

Tells you that the third FASTA file, we count from 0 onwards in the input file, appears 2344 times in the dataset.

4.9.4 Implementation

Finding unique sequences is implemented in *dataset.c* while the interface is implemented in *sequence_multiplicity.c*.

4.10 lengths_from_fasta

The *lengths_from_fasta* tools prints the lengths of sequences stored in a FASTA file.

4.10.1 Usage

The *lengths_from_fasta* tool is to be called like:

```
lengths_from_fasta [fasta]
```

Listing 4.22 – Calling the *lengths_from_fasta* tool

where the *fasta* argument is a FASTA file to print out the lengths of sequences from.

4.10.2 Implementation

The code is implemented in *lengths_from_fasta.c*.

4.11 compare_norms

4.11.1 General

The compare norms utility calculates distances between sequences in a dataset using the Smith Waterman distance and the L_1 and L_2 distance of k-mer frequency vectors that have been projected on their principal components.

4.11.2 Usage

The *compare_norms* tool compares distances between all sequences in a fasta file in Smith Waterman distance and in a subspace yielded by PCA.

```
compare_norms [fasta] [pca] [dimensions] > [output]
```

Listing 4.23 – Calling the *compare_norms* tool

where the arguments are:

1. *fasta* A fasta file containing the sequences to calculate the distances in between them.
2. *pca* A corresponding projection onto principal components from k-mers as obtained by the *kmer2pca* tool outlined in section 4.2
3. *dimensions* The number of projections per sequence to be found in the *pca* file.
4. *output* A file where to output the results.

4.11.3 Algorithm

The Smith Waterman algorithm is used in the same way as pointed out in section 4.4.3. The distance based on the projections is calculated in L_1 and L_2 norm.

4.11.4 Example

```
compare_norms test.fasta test.pca 7 > distances
```

Listing 4.24 – Example of the *compare_norms* tool

The *compare_norm* tool calculates the distances between the sequences within the *test.fasta* in Smith Waterman distance and in L_1 and L_2 norm in the 7 dimensional PCA subspace.

The distances file contains, per line as first and second integers the index of the sequences compared. The third position the Smith Waterman distance, the fourth distance position is the distance in the PCA subspace in L_1 norm and the fifth in L_2 norm. A sample output of looks like this:

```
i j SW L1 L2
1 0 82.000000 2.546303 6.056748
2 0 122.000000 5.428422 10.829566
2 1 84.000000 4.875662 10.343906
3 0 96.000000 4.237413 8.629056
3 1 88.000000 4.427232 8.065469
3 2 118.000000 5.210451 12.005179
4 0 80.000000 3.201523 7.927563
4 1 72.000000 3.526202 7.919044
4 2 106.000000 6.843701 14.724481
```

Listing 4.25 – Example output of the *compare_norms* tool

4.12 adaptive_clustering_X

4.12.1 General

The adaptive clustering tools are at the heart of the MNHN-Tree-Tools suite. These tools run the DBSCAN [2] algorithm with different ϵ parameters creating a layered set of partitions of a dataset to be clustered. These layered structure of clusters allows us to build trees, and gain further insights into a dataset of sequences. Just as the DBSCAN clustering tools (c.f. section 4.4), the adaptive clustering algorithm exists in different variants, corresponding to sequence-distance measure and underlying machine hardware. The tools are: *adaptive_clustering_PCA*,

adaptive_clustering_kmer_L1, *adaptive_clustering_kmer_L2*, *adaptive_clustering_SW_GPU*, *adaptive_clustering_SW_MPI_GPU*

4.12.2 Usage

One can call the utilities with the following commands:

```
adaptive_clustering_PCA [fasta] [initial-epsilon] [delta-epsilon] \
   [minPoints] [split-sets] [n-threads] [dimensions] [pca-file] > [outfile]

adaptive_clustering_kmer_L1 [kmers] [initial-epsilon] [delta-epsilon] \
   [minPoints] [split-sets] [n-threads] > [outfile]

adaptive_clustering_kmer_L2 [kmers] [initial-epsilon] [delta-epsilon] \
   [minPoints] [split-sets] [n-threads] > [outfile]

adaptive_clustering_SW [fasta] [initial-epsilon] [delta-epsilon] \
  [minPoints] [split-sets] [n-threads] > [outfile]

adaptive_clustering_SW_GPU [fasta] [initial-epsilon] [delta-epsilon] \
   [minPoints] [split-sets] [n-threads] > [outfile]

adaptive_clustering_SW_MPI_GPU [fasta] [initial-epsilon] [delta-epsilon] \
   [minPoints] [split-sets] [n-threads] > [outfile]
```

Listing 4.26 – Calling the *adaptive_clustering_X* tools

which have the following arguments:

1. *fasta* The FASTA file containing the dataset to be clustered adaptively and to have a tree built from.
2. *kmers* A k-mer representation of a dataset to be clustered adaptively and to have a tree built from. Such a set can be obtained by the *fasta2kmer* tool highlighted in section 4.1
3. *initial-epsilon* The initial epsilon to start an adaptive clustering run from.
4. *delta-epsilon* The increase in epsilon between successive calls of the DBSCAN algorithm.
5. *minPoints* The minimum number of sequences necessary in an epsilon neighborhood to form or expand a cluster.
6. *split-sets* A *split-set* is a binary file to hold an entire clustering result. As the algorithm creates multiple *split-sets* (one for each stage of the tree) a path with a named prefix has to be provided. The splitsets are named with the given prefix and a number indicating the corresponding tree level, starting at 0 for the outermost leaves of the tree counting upwards until only a single cluster can be formed, the one with the lowest density.

7. *n-threads* The number of threads this run is allowed to use. MPI_GPU runs will fail if this value is not set to 1.
8. *outfile* The outfile of the adaptive clustering run which is valuable not only for informative purposes but also for further tools of this software suite.

4.12.3 Algorithm

The algorithm resides on the DBSCAN algorithm already outlined in section 4.4. The algorithm starts at a given sequence density defined by the parameters *initial-epsilon* and *minPoints*, as highlighted by equation 4.11. Using a successive increase of ϵ by *delta-epsilon* and hence, successive DBSCAN runs several clusterings with ever decreased densities are generated. Clusterings which contain a higher or the same number of clusters as the previous DBSCAN runs are discarded, only clustering results from a run with less clusters at a lower density limit are kept, until a system with a single cluster is reached. We further remark that a result can be obtained at any specified density using the *cluster_dbscan_X* tools highlighted in section 4.4. We shall also outline that this forced monotony in an every decreasing number of clusters further highlights the importance of a wisely chosen starting point defined by: *initial-epsilon* and *minpoints*. The starting point shall be chosen at a density where the highest number of clusters is suspected.

From the multiple clustering results at different densities, as obtained by this tool, we can infer a hierarchy by comparing clusters of different layers and verifying if assumed smaller clusters at higher density limits are a part of larger clusters in results from clustering runs at lower density limits. Connections between clusters are created and print to the output file if a cluster from with higher density limit is contained with at least 80% of its sequences in a cluster with a lower density limit. The inner workings of the algorithm are outlined in figure 4.2 The algorithm inherits the same optimizations as those pointed out in section 4.4.3. Further the algorithm is parallelized precalculating layers at different ϵs in different threads. Finally a Message Passing Interface (MPI) version of the GPU based Smith Waterman DBSCAN region expand function exists. In this function the distance between a sequence in question with all other sequences of

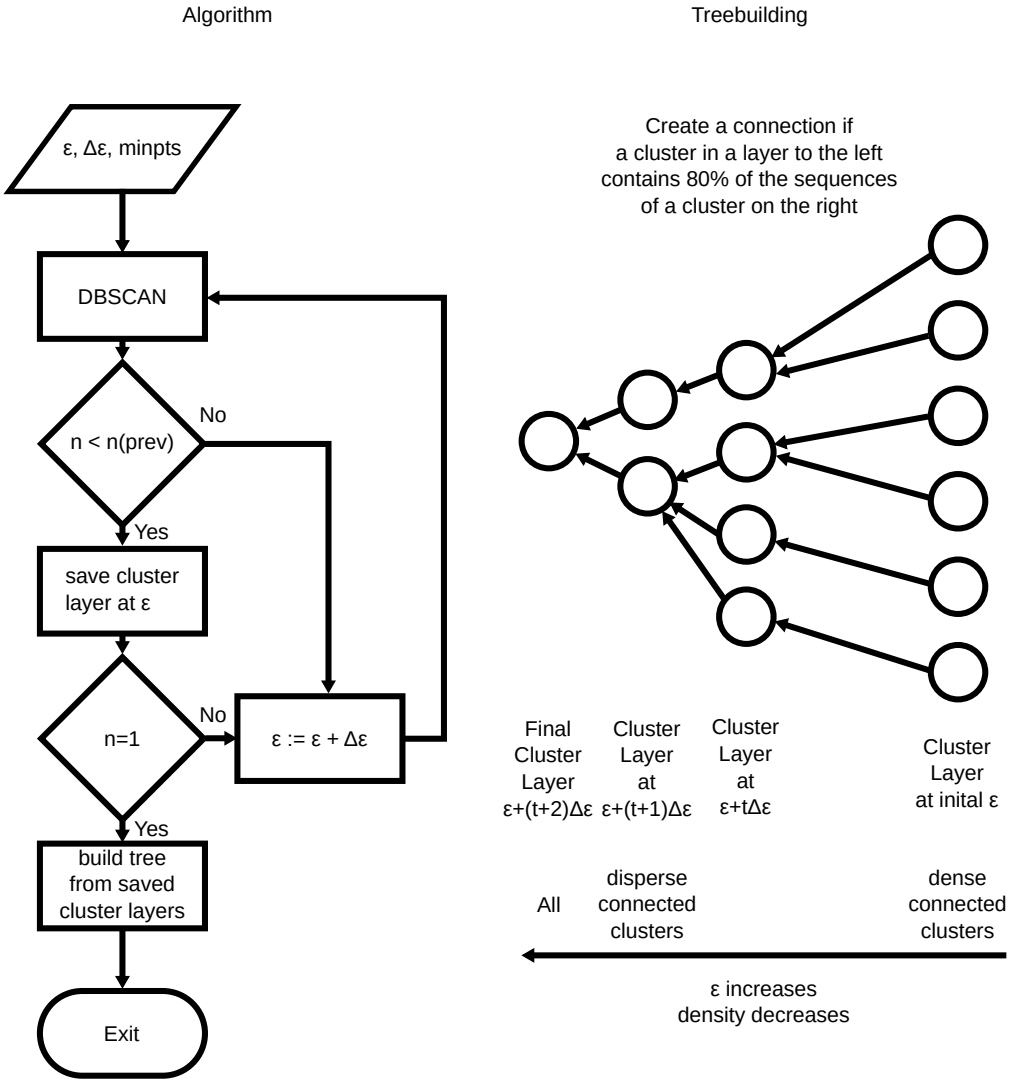


Figure 4.2 – **Building trees from sequences:** Highlights our algorithm, and its three parameter input ϵ the initial epsilon neighborhood radius for the DBSCAN algorithm, $\Delta\epsilon$ the increase of ϵ in every step and minpts the minimal number of points to be found in an epsilon neighborhood to either extend or create a cluster. n in the diagram represents the number of clusters of the current DBSCAN run, $n(\text{prev})$ the number of clusters of the previous run. t at the treebuilding side, the stepnumber and hence, how often DBSCAN has been run. The algorithm works itself from the right to the left through the tree, detecting first using a small ϵ very dense clusters of nucleic sequences and hence, clusters that are highly conserved in sequence. In increasing DBSCANS ϵ from layer to layer, the clusters contain less and less conserved sequences, and basically fusion from layer to layer, until DBSCAN just detects a single cluster, the root of the tree.

the dataset is to be evaluated in order to know if these distances are smaller than ϵ or not. Using the MPI interfaces the algorithm has the possibility to profit from multiple available GPUs in multiple machines in a clustering environment. We have successfully tested this algorithm on six independent cluster nodes with four GPUs each using an infiniband network backbone.

4.12.4 Example

```
adaptive_clustering_PCA test.fasta 0.02 0.01 4 /tmp/out-splits- 8 7 test.pca
```

Listing 4.27 – Calling the *adaptive_clustering_PCA* tools

The command performs an adaptive clustering run using the sequences stored in test.fasta by applying DBSCAN on the 7 dimensional k-mer subspace defined by the projections onto the principal components stored in test.pca. The *inital-epsilon* is defined to be 0.02, the *delta-epsilon* is defined to be 0.01 and a minimum number of points to be found within an epsilon neighborhood for cluster expansion or creation is defined to be four. The resulting clusters for each layer are stored in binary form at */tmp/out-splists-X* where X counts from 0 to the number of layers found until only a single cluster is retrieved due to the small density at a large ϵ.

4.12.5 Implemenation

The algorithm and its interfaces are implemented in *adaptive_clustering.c*. The code makes use of the DBSCAN algorithm implemented in *dbscan.c*.

4.13 split_set_to_fasta

This tool creates FASTA files for clusters found in a *split-set* stored in binary form like the ones generated by the *adaptive_clustering_X* tools, highlighted in section 4.12.

4.13.1 Usage

The *split_set_to_fasta* tool can be called in the following way:

```
split_set_to_fasta [fasta] [split-set] [out-path-name]
```

Listing 4.28 – Calling the *split_set_to_fasta* tool

with the arguments:

1. *fasta* A FASTA file containing the whole dataset from that is partitioned by a *split-set*.
2. *split-set* The binary *split-set* containing the clusters.
3. *out-path-name* The path and name of the resulting FASTA files that are suffixed with unique number for each cluster in stored in the *split-set*

4.13.2 example

```
split_set_to_fasta test.fasta /tmp/s0023 /tmp/fa0023-
```

Listing 4.29 – Example of the *split_set_to_fasta* tool

Here the *split-set /tmp/s0023* is written to individual FASTA files per clusters stored to */tmp/fa0023-X* where X is a number for each individual cluster. *test.fasta* holds the original dataset which is needed as the *split-set* file only stores indices and not the sequences itself.

4.13.3 Implementation

The tool is implemented in *split_set_to_fasta.c*.

4.14 split_set_to_projections

The *split_set_to_projections* tool allows you to get clusterwise principal component projections from the clustering information stored in binary *split-set* files obtained by an adaptive clustering run on a dataset, i.e. obtained from the tool shown in section 4.12 and a PCA projection of the same dataset that can be obtained using the *kmer2pca* tool presented in section 4.2. This tool is especially useful to plot individual clusters together with a PCA analysis of sequences and hence, to find sequences corresponding to clusters in such a diagram.

4.14.1 Usage

The *split_set_to_projections* tool is to be called like:

```
split_sets_to_projections [fasta] [pca] [dimensions] [split-set] [projections]
```

Listing 4.30 – Calling the *split_set_to_projections* tool

with the following arguments:

1. *fasta* The FASTA file that is partitioned by clusters stored in the *split-set*.
2. *pca* A projection onto principal components of the affor-mentioned FASTA file, i.e. obtained with the *kmer2pca* tool mentioned in section 4.2.
3. *dimensions* The number of dimensions of the PCA subspace stored in the afformentioned *pca* file.
4. *split-set* The binary file that holds the clusters.
5. *projections* A *path/filename* combination where to put the resulting projections of the clusters in PCA space. Each cluster will generate a file with an individual number added to the name.

4.14.2 Example

```
split_sets_to_projections test.fasta test.pca 7 /tmp/s0023 /tmp/s0023proj
```

Listing 4.31 – Calling the *split_set_to_projections* tool

In this case clusters stored in */tmp/s0023* a binary *split-set* file are searched in the 7 dimensional subspace projection onto the principal components of a k-mer representation of the sequences found in the *test.fasta* FASTA file. The projections of each cluster are stored in a separate file */tmp/s0023projX* where X is a number corresponding to each cluster.

4.15 split_sets_to_newick

This tool allows you to create a Newick [7] tree file from the *split-set* and *output* files that have been generated by an adaptive clustering run as outlined in section 4.12.

4.15.1 Usage

The tool can be used as outlined in the following:

```
split_sets_to_newick [filter] [lengths-from-output] [dimensions] \
  [outfile] [minPoints] [file1 ... filen] > [newick]
```

Listing 4.32 – Calling the *split_sets_to_newick* tool

with the arguments:

1. *filter* One may use 0, for no filter or any other number to define the minimum cluster size to occur in the resulting tree. Filtering the tree in this way changes the labels on the tree. In order to retain correct labels we suggest to use the *filter_split_sets_by_min* tool, shown in section 4.19, prior using this tool and to apply no filter at this step. This parameter is kept for convenience.
2. *lengths-from-output* Allows us to define distances between nodes of the Newick tree utilizing the parameters found in the output file of an adaptive clustering run performed with one of the tools shown in section 4.12. If you do not choose to tree lengths proportional to density differences between layers choose 0 here, 1 if your run uses an L_1 based distance calculation and 2 if your run uses an L_2 based distance calculation. 1 uses a hypercube with a sidelength of 2ϵ to calculated the volume, 2 uses a hypersphere with radius ϵ.
3. *dimensions* As the lengths between the nodes correspond to density differences we need the dimensions in order to be able to calculate the volume using the different ϵ radiuses found in the *outfile* file. As a basic guidance choose the number of projections onto principal components that you have retained in case this run was performed with PCA data. Otherwise choose 1 or a value to your convenience.
4. *outfile* The *output* file from an adaptive clustering run generated from one of the tools outlined in section 4.19.
5. *minPoints* The *minPoints* argument used during the adaptive clustering run which is needed in order to correctly calculate the correct density differences.
6. *file1*...*filen* All the *split-set* files to use to built this tree from. In order to select them in general wildcards can used on shells like the bash by i.e. using something like

   ```
   /path/prefix*
   ```

   .
7. *newick* The created Newick tree file.

The arguments *dimensions*, *outfile* and *minPoints* are to be omitted if 0 was chosen for the *lengths-from-output* argument.

4.15.2 Algorithm

If the lengths in the newick file are chosen to be density differences, the density is calculated from the ϵ value in the output file of an adaptive clustering run. The formulas used for the volume are:

$$V = (2\epsilon)^d, \tag{4.19}$$

for the hypercubic L_1 distance case, and

$$V = \epsilon^d \frac{\pi^{d/2}}{\Gamma(d/2+1)} \tag{4.20}$$

for the hyperspherical case. Here d is the number of dimensions and Γ is the Γ-function.

The algorithm builds the Newick file using a recursive function traversing the nodes. Connections between nodes in different layers are made if from a layer corresponding to denser clusters a cluster is contained at least at 80% in a layer corresponding to more diffuse clusters.

4.15.3 Example

```
split_sets_to_newick 0 2 7 outfile 4 clusters/out-* > tree.dnd
```

Listing 4.33 – Example of the *split_sets_to_newick* tool

Here we generate a Newick tree that we store in the *tree.dnd* file. The lengths of the tree are density difference based, from an adaptive clustering run whose output has been stored in *outfile* and whose *split-sets* have been stored to *clusters/out-X* files. A *minPoints* argument of 4 was used during the adaptive clustering run, which was PCA based and operated on a seven dimensional subspace.

4.15.4 Implementation

The interface for Newick tree generation is implemented in *split_sets_to_newick.c*. The algorithms that allow us to generate a tree are found in *cluster_io.c*.

4.16 split_set_to_matrix_X

This are two tools to create a matrix view of a tree that can be visualized with the *gnuplot* [5] program. The matrix view highlights each sequence as a point on a line, coloring sequences belonging to the same cluster in the same color. This is useful for instance in the verification of simulated or real datasets where the sequences are already ordered by clusters and hence should generate lines in the same color. This representation also allows one to investigate how clusters merge as one stacks these lines on top of each other as the ϵ value increases.

4.16.1 Usage

The tools *split_set_to_matrix_annotation* and *split_set_to_matrix_line* can be used together to form a readable output file.

```
split_set_to_matrix_annotation [fasta] >> [matrix]
split_set_to_matrix_line [fasta] [split-set] >> [matrix]
```

Listing 4.34 – Calling the *split_set_to_matrix* tools

where:

1. *fasta* Is a FASTA file that one has generated clusters from, for instance in performing an adaptive clustering run using the tools outlined in section 4.12.
2. *split-set* A binary file holding the clusters obtained from an adaptive clustering run.
3. *matrix* The matrix to store a line, correponding to the *split-set* file, to.

The *split_set_to_matrix_annotation* call in itself is not needed, it basically annotates each sequence with the internal sequence number, or the number of sequence occurrence in the original FASTA file, starting with 0 for the first sequence.

4.16.2 Example

We imagene we have performed an adaptive clustering run that yielded 86 different cluster layers (of a tree) and hence, 86 *split-set* files in using one of the adaptive clustering tools shown in section 4.12

```
for((i=0;i<85;i++))
do split_set_to_matrix_line test.fasta /tmp/out`printf %04d $i` >> test.matrix
done
```

Listing 4.35 – Example usage of the *split_set_to_matrix* tools

In this example the 86 *split-sets* for each layer of the adaptive clustering run are successively as lines written to the *test.matrix* file. The *test.matrix* file can then be visualized by *gnuplot* using the following command:

```
plot 'test.matrix' matrix with image
```

Listing 4.36 – Using gnuplot to plot a matrix from *split_set_to_matrix* tools

to obtain a map with different colors for each cluster.

4.17 split_set_from_annotation

4.17.1 General

This tool allows you to generate binary clustering sets such as those created using by an adaptive clustering run by hand, annotating each cluster.

4.17.2 Usage

The tool can be used like:

```
split_set_from_annotation [fasta-ds] [annotation] [unique-annotations] \
  [split-set]
```

Listing 4.37 – Calling the *split_set_from_annotation* tool

with the following arguments:

1. *fasta-ds* A FASTA file that shall be partitioned according to the annotations in the *annotation* file.
2. *annotation* The annotation file, containing a line for each sequence in the *fasta-ds* file. An annotation can be any chosen word. If a line contains only a dash '-', the sequence will not be part of any of the possible clusters.
3. *unique-annotations* The database of possible annotations. For each cluster a single word used to annotate sequences in the *annotation* file corresponding to this cluster is made. The entries shall be made in the order that you wish to have your clusters in the resulting binary *split-set* file.

4. *split-set* The resulting binary clustering or *split-set* file.

4.17.3 Example

We imagine a fasta file:

```
>sequence1
AACCTT
>sequence2
AGCCTT
>sequence3
TTTTTT
>sequence4
AAACTT
>sequence5
AAACGT
>sequence6
AAAGCT
```

Listing 4.38 – Example of a dataset clustered with the *split_set_from_annotation* tool

and we want to partition this file into two clusters, one containing sequence 1, and sequence 2 while the second cluster is composed of sequence 4, 5 and 6. We would like to omit sequence 3 from the partitions. As such we create the following annotations file:

```
one
one
-
two
two
two
```

Listing 4.39 – Example of an *annotation* file for the *split_set_from_annotation* tool.

As shown we annotate all sequences of the first cluster with one, and all sequences of the second cluster with two. We omit the third sequence by annotating it with a dash '-'. Further we need to write a file containing all the words that we used to annotate our clusters.

```
one
two
```

Listing 4.40 – Example of an *unique-annotations* file for the *split_set_from_annotation* tool.

With the files at hand we can call the tool:

```
split_set_from_annotation test.fasta annotation unique-annotations /tmp/out-set
```

Listing 4.41 – Example of the *split_set_from_annotation* tool

and obtain the binary *split-set* clustering for the clustering at our wish in */tmp/out-set*.

4.17.4 Implementation

The tool is implemented in *split_set_from_annotation.c*.

4.18 split_set_from_swarm

4.18.1 General

This tool generates a binary *split-set* file describing clusters in *MNHN-Tree-Tools* format from a clustering obtained with Swarm version 3 [12].

4.18.2 Usage

The tool can be used in the following manner:

```
split_set_from_swarm [fasta] [swarm] [split-set]
```

Listing 4.42 – Calling the *split_set_from_swarm* tool

with the following arguments:

1. *fasta* The original FASTA file that has been clustered with Swarm version 3 [12]. This is not the downsampled file that is used to perform clustering with the swarm tool, but the original dataset. (c.f. example).
2. *swarm* The clustering performed by swarm. Swarms output file
3. *split-set* The binary file containing the clustering to be created.

4.18.3 Example

We imagine we would like to generate a clustering of several sequences with swarm. Hence, we first have to eliminate duplicates and format sequence multiplicity in a way that swarm understands it. The *sequence_mulitplicity* tool, outlined in section 4.9, is perfect for this task. As such we prepare the swarm input file as follows:

```
sequence_multiplicity test.fasta > swarm-in.fasta
```

Listing 4.43 – Generating a valable input file for swarm using the *sequence_multiplicity* tool

We than run the swarm tool to cluster our dataset.

```
swarm swarm-in.fasta > swarm-output
```

Listing 4.44 – Calling Swarm before the *split_set_from_swarm* tool

and can finally built a *split-set* cluster file that uses the interal structures of MNHN-Tree-Tools.

```
split_set_from_swarm test.fasta swarm-output /tmp/split-set
```

Listing 4.45 – Example of the *split_set_from_swarm* tool

Note that *test.fasta* is used in the final call and not *swarm-in.fasta*. The tool can by tracking the correct annotations find the correct sequences in the original dataset and built a binary *split-set* file accordingly.

4.18.4 implementation

This tool is implemented in *split_set_from_swarm.c*.

4.19 filter_split_sets_by_min

4.19.1 General

This tool allows one to filter clusters obtained from an adaptive clustering run, as outlined in section 4.12, that contain less sequences than a certain threshold, and to retain clusters that contain more sequences than this threshold.

4.19.2 Usage

The tool is to be called like:

```
filter_split_sets_by_min [minimum] [filtered-split-sets-path-name] \
   [input-split-sets]
```

Listing 4.46 – Calling the *filter_split_sets_by_min* tool

where the arguments are:

1. *minimum* An integer representing the minimum number of sequences a cluster has to contain in order to be retained in the dataset
2. *filtered-split-sets-path-name* A path and name where the filtered *split-sets* should be stored.
3. *input-split-sets* The unfiltered *split-sets* to filter using this algorithm.

4.19.3 Example

```
filter_split_sets_by_min 100 /tmp/f100s /tmp/s*
```

Listing 4.47 – Example of the *filter_split_sets_by_min* tool

Here, the program reads the *split-sets*, clusters of each layer, and filters clusters that contain less then 100 sequences from the dataset. The new dataset without these clusters is written to */tmp/f100sX* where X is the number for each layer in the output *split-set*. The selection using the wildcard works as the original *split-set*s are named */tmp/sX* and the numbers X are created in a way by the adaptive clustering tools such that the wildcard selection keeps them in correct order.

4.19.4 implementation

The code and the interface is implemented in *filter_split_set_by_min.c*. The filtering mechanism in *cluster_io.c*

4.20 find_sequence_in_split_sets

4.20.1 General

The tool allows you to find a specific sequence among the clusters obtained from an adaptive clustering run as performed, for instance, with the tools outlined in section 4.12.

4.20.2 Usage

The utility can be called like:

```
find_sequence_in_split_set [fasta-ds] [fasta-seq] [split-sets]
```

Listing 4.48 – Calling the *find_sequence_in_split_sets* tool

with the arguments:

1. *fasta-ds* A FASTA file containing the entire dataset of sequences that an adaptive clustering run was performed on.
2. *fasta-seq* A FASTA file containing a singe sequence, that should be searched for in the obtained clusters
3. *split-sets* A list of binary files, that represent the different layers of clusters, obtained from an adaptive clustering run.

4.20.3 Example

```
find_sequence_in_split_set test.fasta seq.fasta /tmp/s*
```

Listing 4.49 – Example of the *find_sequence_in_split_sets* tool

The command shown above searches for the sequence defined in *seq.fasta*, which has to be part of the complete dataset *test.fasta*, in the different *split-sets* selected with */tmp/s**. The wildcard here, as with most tools that work with multiple *split-sets*, will work as the files are numbered correctly for such a selection during an adaptive clustering run. The tool prints an output like the following:

```
Sequence has internal ids:
0
Sequence is found in Clusters
L22C0 1 times
L23C0 1 times
L24C0 1 times
L25C0 1 times
L26C0 1 times
L27C0 1 times
```

Listing 4.50 – Output of the *find_sequence_in_split_sets* tool

The designation of 0 as internal id means that in FASTA outputs of the clusters, such as those created with the tool *split_set_to_fasta* (c.f. section 4.13), the sequence is named *>sequence_0* as it is, in this case, the first sequence of the original dataset. The sequence is found in clusters named *LXCY*. Where X represents the layer number and C the cluster number. As such the sequence is to be found in the binary *split-set* bearing the number X at its end. Once this binary *split-set* is converted to clusters, i.e in FASTA form, the sequence is found in the files with the suffix Y. One may also say that in the tree built from such an adaptive run the sequence is to be found in the tree layer X in cluster Y.

4.20.4 Implementation

The interface as well as the tool itself is implemented in *find_sequence_in_split_sets.c*.

4.21 print_connections

The *print_connections* tool allows you to highlight connections in the same way as they are listed in the output file of an adaptive clustering run. This means it highlights which cluster of a layer with a higher sequence density limit obtained from an adaptive clustering run is embedded into a cluster of a layer with a lower squence density limit. A connection is formed if 80% of the sequences of a cluster of a higher density is found in a cluster with a lower density limit.

4.21.1 Usage

The tool can be called issuing the folling command:

```
print_connections [lower-layer-split-set] [upper-layer-split-set]
```

Listing 4.51 – Calling the *print_connections* tool

where *lower-layer-split-set* and *upper-layer-split-set* are binary clusterings obtained from an adaptive clustering run that was performed with the tools outlined in section 4.12. The *upper-layer-split-set* should be a clustering with a higher density then the *lower-layer-split-set*.

4.22 tree_map_for_sequence

This tools allows you to locate a certain sequence in a Newick tree. This tool creates a *map* file that you can use in conjunction with the Newick utilities [8] published by the University of Geneva, in order to highlight clusters in different colors that contain such a certain sequence.

4.22.1 Usage

The *tree_map_for_sequence* tool can be used as outlined in the following:

```
tree_map_for_sequence [fasta-ds] [fasta-seq] [color] [split-sets] > [map]
```

Listing 4.52 – Calling the *tree_map_for_sequence* tool

with the arguments:

1. *fasta-ds* A FASTA file containing the full dataset of that has been used to perform an adaptive clustering run, for instance, with the tools shown in section 4.12.
2. *fasta-seq* A FASTA file containing the single sequence to be highlighted in a Newick tree by using this tool in conjunction with the Newick utilities.
3. *color* A string defining the color to use to mark clusters that contain the sequence in question. The color has to be expressed in a Cascade Style Sheet (CSS) compatible way. Hence, simple color expressions such as red, green and blue should work. In general it is preferred to use colors in the format #RRGGBB where lead by a hash three hexadecimal numbers define the color values for red, green and blue. In order to get an orange color you may use, #FF8800 which corresponds to full red, half green and no blue intensity.
4. *split-sets* All the binary clusterings that are building the tree and that are used running the *split_sets_to_newick* tool, outlined in section 4.15, in order to generate the dendogram that the Newick tools will use in the final tree visualization.
5. *map* The resulting *map* file that can be used together with the Newick utilities to create a tree where all clusters containing a specific sequence are colored.

4.22.2 Example

```
tree_map_for_sequence test.fasta seq.fasta '#FF0000' /tmp/s* > tree.map
```

Listing 4.53 – Example of the *tree_map_for_sequence* tool

In this example we generate a map file *tree.map* to color all clusters that contain the sequence contained in *seq.fasta* red in a tree that we create using the Newick utilities. The tree was built from the complete dataset of sequences found in *test.fasta*. The binary *split-sets* that build the different layers of the tree are found at

/tmp/sX where X is the number of each layer. As with all the previous tools the wild card selection should work perfectly. To better illustrate the usage of the tool we show in the following listing how this tool is used together with the the *split_sets_to_newick* tool described in section 4.15, and the Newick utilities [8] by the university of Geneva:

```
split_sets_to_newick 0 test.fasta 0 /tmp/s* > tree.dnd
tree_map_for_sequence test.fasta seq.fasta '#FF0000' /tmp/s* > tree.map
nw_display -w 600 -rs -i 'opacity:0' -b 'opacity:0' -l 'opacity:0' \
  -c tree.map tree.dnd > tree.svg
```

Listing 4.54 – Further example of the *tree_map_for_sequence* tool

where *tree.dnd* is the Newick file that corresponds to the *map* *tree.map* that we generate with the tool outlined in this section. The *nw_display* from the Newick utilities [8] creates a (Scalable Vector Graphics) SVG image of our tree without annotations and a width of 600 points in a radial fashion. It uses the aforementioned *map* to color clusters that contain the sequence to be highlighted red in the tree. An example tree generated like this is shown in figure 4.3.

4.22.3 Implementation

The *tree_map_for_sequence* tool is implemented in *tree_map_for_sequence.c*.

4.23 tree_map_for_split_set

The *tree_map_for_split_set* tool allows you to highlight clusters in trees in different ways. First you can identify different clusters in a tree. Second you can visualize how many sequences of certain cluster propagate through different clusters in the tree. Finally this tool can also be used in conjunction with different graphics software packages to build more complex trees by overlaying multiple representations that were generated with the help of this tool. This tool generates *map* files that you can use together with the Newick tools published by the university of Geneva [8].

4.23.1 Usage

The tool can be called in the following manner:

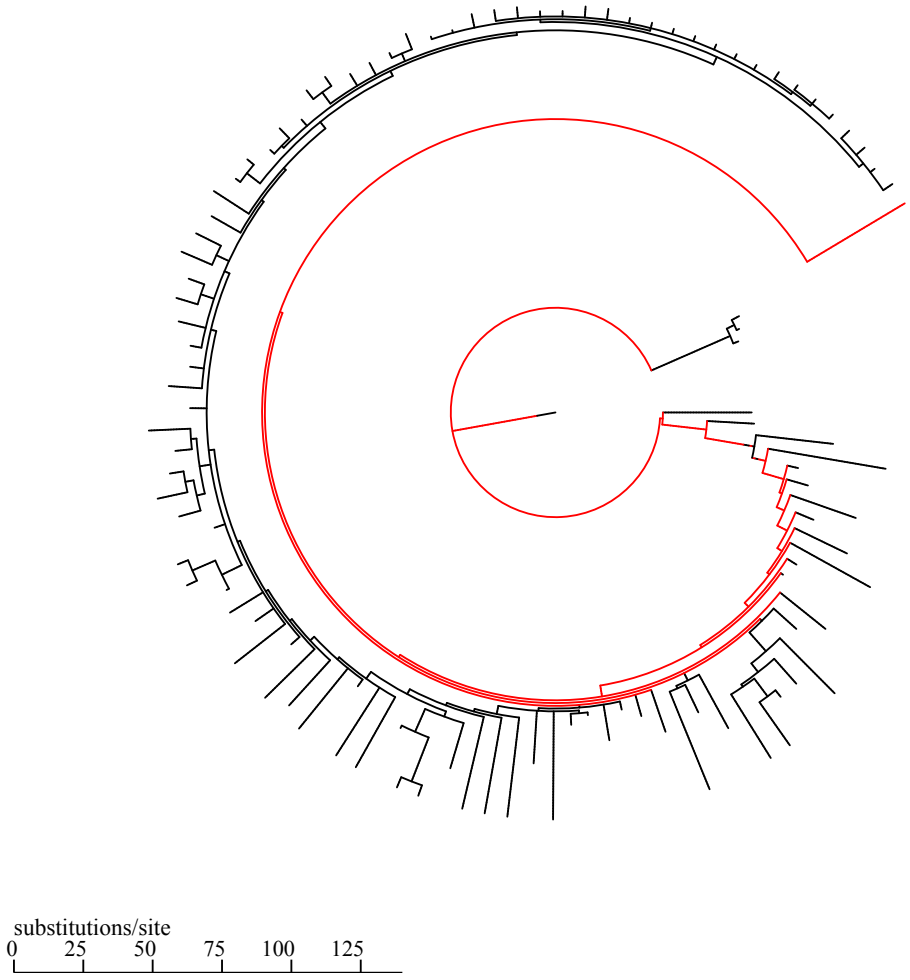


Figure 4.3 – An example of a tree that outlines with a red line the location of a certain sequence. The commands to create such a tree are outlined in listing 4.54. The substitutions/site scale automatically generated by the Newick utilities [8] and can be ignored.

```
tree_map_for_split_set [split-set] [fasta-ds] [number-of-clusters] \
   [gradient-type] [culster-index-colors] [split-sets-full-tree] > [map]
```

Listing 4.55 – Calling the *tree_map_for_split_set* tool

where the arguments are the following:

1. *split-set* The *split-set* to color, highlight on the Newick tree.
2. *fasta-ds* The complete dataset from which an adaptive clustering run has generated clusters and hence, a tree structure.
3. *number-of-clusters* How many clusters to color in the same diagram. Currently coloring more than 1 cluster might result in unpredictable behavior.
4. *gradient-type* Colors the clusters from white to the desired color (by number of sequences of the clusters found in a cluster, 0 sequences for white and all, the color that you have chosen). The possible values to select are:
   — 0: no gradient and thus all clusters contain at least 1 sequence are colored in the desired color.
   — 1: a linear gradient from white corresponding to *no sequences*, to the desired color corresponding to all sequences.
   — 2: la logarithmic gradient. The logarithmic gradient sets 0 sequences to all white and colors clusters containing sequences of the cluster to be highlighted by the number of sequences contained of the cluster to be highlighted in a logarithmic fashion.
5. *cluster-index-colors* Allows you to select the different colors for different clusters. As already mentioned this might result in unpredictable behavior selecting more than 1 cluster and more than one color. This argument has to be formed as a set of strings like *'2,#FF0000' '4,#00FF00'* where the first number is the number of the cluster to select from the *split-set* which might hold n clusters. This number is followed by a comma "," and a color value, expressed by 3 hexadecimal number, with the first two being the red, the second two to the green and the third pair the blue component.
6. *split-sets* The clusters as obtained by an adaptive clustering run with the tools mentioned in section 4.12.

7. *map* The resulting *map* file that can be used together with the Newick utilities to visualize a tree with colored clusters containing sequences of the *split-set* according to the rules chosen above.

4.23.2 Example

```
tree_map_for_split_set cluster_set test.fasta 1 2 '5,#000000' /tmp/s* > tree.map
```

Listing 4.56 – Example of the *tree_map_for_split_set* tool

In this example the tool takes the 5th cluster from the binary cluster file *cluster_set* that contains partitions of the sequence dataset found in *test.fasta*. The tool generates a *tree.map* file for a dendogram that was built from all the binary cluster files stored at */tmp/sX*, where X corresponds to the layer number. In this case the map is built in such a way that only clusters containing sequences from the 5th cluster of the *cluster_set* are shown in a logarithmic gradient, from white (no sequences) to black, #000000, corresponding to clusters holding all sequences.

Even though this tool was at one point designed to permit different clusters to be highlighted in different colors, subsequent modifications of this tool destroyed this functionality and selecting multiple clusters with different colors might result in unexpected behavior. Therefore we recommend you to limit yourself to highlight only one cluster per dodendogram and use different image processing software tools to overlay resulting trees.

An example for generating a tree in conjunction with other tools might look like the following:

```
split_sets_to_newick 0 test.fasta 0 /tmp/s* > tree.dnd
tree_map_for_split_set cluster_set test.fasta 1 2 '5,#000000' /tmp/s* > tree.map
nw_display -w 600 -rs -i 'opacity:0' -b 'opacity:0' -l 'opacity:0' \
  -c tree.map tree.dnd > tree.svg
```

Listing 4.57 – Extended example of the *tree_map_for_split_set* tool

where we use the *split_sets_to_newick* tool (c.f. section 4.15) to generate a Newick file, and the *tree_map_for_split_set* tool shown in this section to generate a corresponding map, highlighting as explained above only sequences from the 5th cluster of the *cluster-set* binary file. We than use *nw_display* from the Newick tools [8] to visualize the final dendrogram in a SVG file.

A diagram generated with a logarithmic gradient might look like the one shown figure 4.4

4.23.3 Implementation

The *tree_map_for_split_set* tool is implemented in *tree_map_for_split_set.c*.

4.24 virtual_evolution

The tools *virtual_evolution* and *virtual_evolution_controlled* allow you to simulate simple evolutionary processes and create nucleic sequences ressembling to evolutionary processes. Your simulated biosphere shall hence reside in a FASTA file.

4.24.1 Usage

The evolution simulation as implemented in MNHN-Tree-Tools comes in two versions: A rather crude *virtual_evolution* tool and in a more controlled, later added *virtual_evolution_controlled* tool.

The tools can be called as outlined in the following:

The *tree_map_for_sequence* tool can be used:

```
virtual_evolution [seed] [seq-length] [n-sequences] [n-partitions] \
  [max-time] [rate] [n-mutations-to-amplification-template] [fasta]
virtual_evolution_controlled [seed] [seq-length] [n-sequences] [n-partitions] \
  [max-time] [rate] [n-mutations-to-previous-amplification-template] [fasta]
```

Listing 4.58 – Calling the *virtual_evolution* tools

with the parameters:

1. *seed* The seed the pseudorandom number generator uses. This value allows us to generate the same set of sequences in subsequent calls of this tool.
2. *seq-length* The length of the sequences to be generated.
3. *n-sequences* How many sequences in total have to be generated.
4. *n-partitions* How many families/clusters shall be generated, or in other words, how many amplification events should happen.
5. *max-time* This value selects if the evolution should occur at random times beteween amplifications and be limited to *max-time* or weather, if -1 is chosen a constant evolution

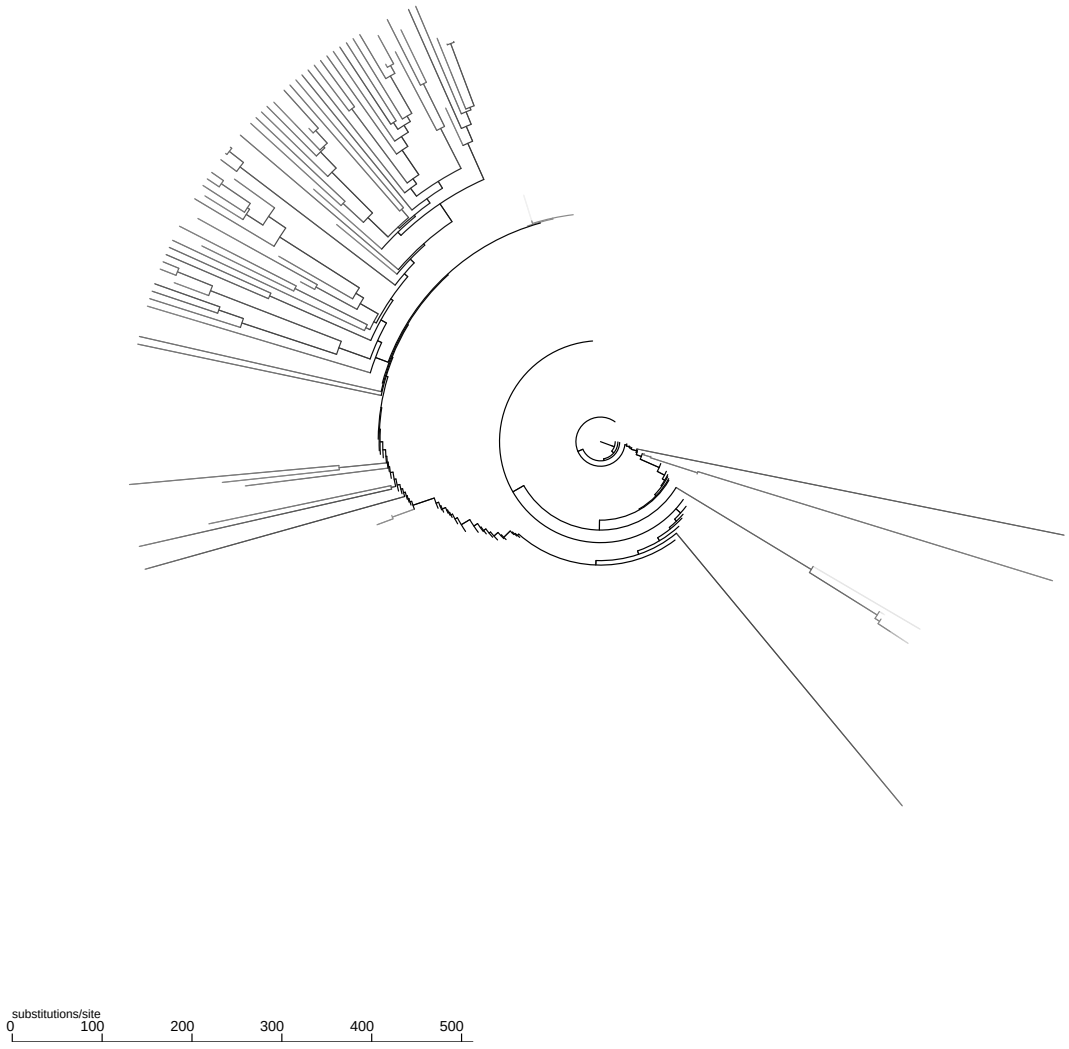


Figure 4.4 – An example of a tree that used the *tree_map_for_split_set* tool to only highlight a certain family in a tree. One may observe the logarithmic gradient as clusters with few sequences are shown in light gray, while clusters containing the entire set of sequences, that belong to the outlined family are drawn dark or black. The substitutions/site are automatically generated by the Newick tools [8] and are to be ignored.

cycle befor a new cluster expansion happens is prefered. If the value selected is not -1 a random number, between 0 and *max-time* of evolution cycles may occur.

6. *rate* The substitution rate of nucleotide bases in the nucleic sequences per evolution cycle.
7. *n-mutations-to-amplication-template* Adds the here chosen number of mutation to the amplification template before a new cluster expansion occurs. In this case the nucleotide bases are only replaced X times, as chosen here. The actual number of mutations might be lower as the replaced bases (at a probablity of 1/4th per base) might be of the same nucleotide as the one that was in place.
8. *n-mutations-to-previous-amplification-template* Adds this number of real mutations to the previous amplification template, before generating a new family.
9. *fasta* A FASTA file where to write the sequences generated to.

4.24.2 Algorithm

As the algorithms for the two simulation processes are different we outline both in the following:

— *virtual_evolution* process. The evolutionary process herein is a basic expansion and diffusion process. The way this simulator works is that it generates a template sequence of a given size. It then copies this template sequence until it fills a whole family as defined by the *n-sequences* and *n-partitions* arguments. A family or partition as such contains

$$N = n_{\text{seq}}/n_{\text{part}}, \tag{4.21}$$

sequences where n_{seq} are the number of total sequences and n_{part} are the number of partitions or families to be created. It than performs

$$M = ptr \tag{4.22}$$

mutations on the previous partitions generated. Where t is a random time, and hence the number of cycles number, which is chosen between 0 and t_{max} which is defined by the

max-time parameter. r is the mutation rate as defined by the *rate* parameter. p is the number of partitions already generated. Once M mutation were perfomed on the current state of the dataset a sequence randomly chosen from the previous sequencies will be the template for the next expension of N sequences. The algorithm is continued until all partitions are created, perfoming a last mutation run on the entire dataset as determined by equation 4.22. The whole process is further outlined in figure 4.5.

— *virtual_evolution_controlled* is a different, simpler but more controlable process. The main difference to the uncontrolled *virtual_evolution_process* is that the template for each family is chosen from the previous template to create a partition or family, and not randomly chosen from the existing sequences. Further the number of mutations is enforced not be just exchanges of nucleotides, that might actually not lead to mutations as there is a 1 in 4 chance that a nucleotide gets replaced by itself, but to be actual mutations. Hence, one perfectly knows to what extend the templates between families differ. Together with a fixed mutation rate one can further perfectly estimate the diffusion of each partition. As such we have created a perfectly controlled distance-diffusion mechanism. The process is outlined in figure 4.6.

4.24.3 Example

```
virtual_evolution_controlled 42 100 10000 10 -1 1000 4 biosphere.fasta
```

Listing 4.59 – An example of how to use the *virtual_evolution* tools.

In this example the evolutionary process generates 10000 sequences of 100 nucleotides, in 10 partitions of 1000 sequences and a mutation rate $r = 1000$. The time $t = 1$, or number of cycles is fixed to one in order for the diffusion to stay constant during the process. The template undergoes 4 mutations before creating each new family. The resulting sequences are stored to *biosphere.fasta*.

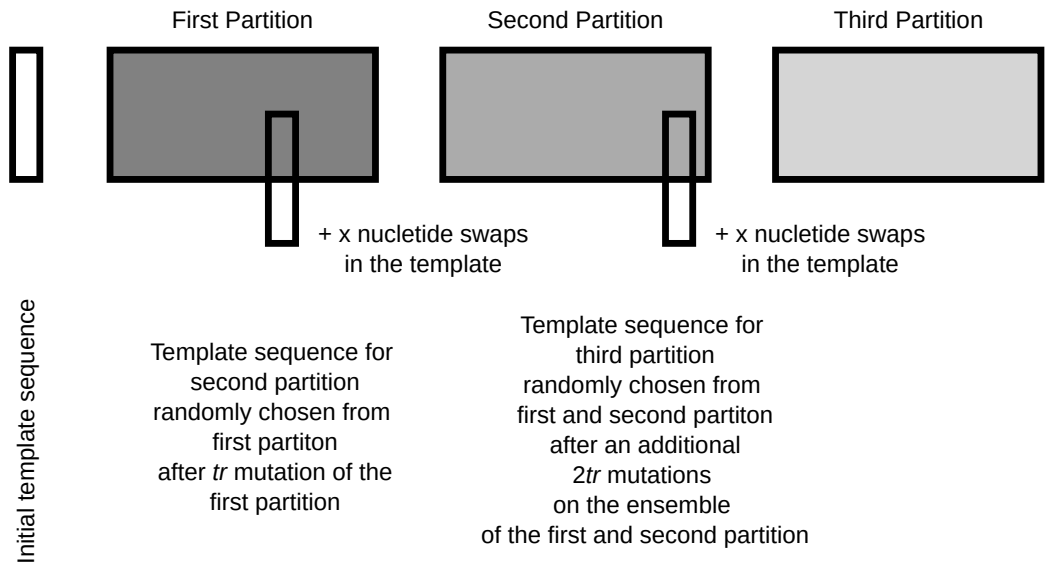


Figure 4.5 – The classic virtual evolution process. The different levels of gray show how diffused, (old) the sequences are at the end of the process. Please note that t is chosen at random after each partition expension.

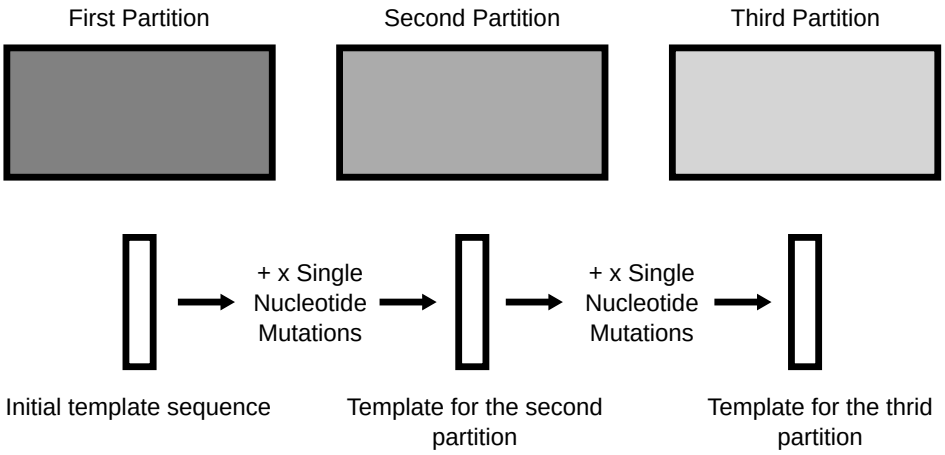


Figure 4.6 – The more restricted controlled virtual evolution process. The different level of gray show how diffused, (old) the sequences are at the end of the process. Again note there are $M = ptr$ mutations happening across the available sequences after each expansion and that t might be variable if chosen so.

4.24.4 Implementation

virtual_evolution and *virtual_evolution_controlled* are implemented in *virtual_evolution.c* and *virtual_evolution_controlled.c* respectivly.

4.25 simulation_verification

4.25.1 General

This is a small tool that tells you how many families created by a virtual evolution run performed using the tools shown in section 4.24 can be found by an adaptive clustering run executed using the tools shown in section 4.12. As such this is an internal quality measurement tool.

4.25.2 Usage

The tool can be run by issuing the following command:

```
simulation_verification [fasta] [partitions] [precision] [split-sets]
```

Listing 4.60 – Calling the *simulation_verification* tool

with the arguments:

1. *fasta* A FASTA file containing a virtual evolution dataset as generated with the tools in section 4.24.
2. *partitions* The number of partitions in the dataset as chosen during the virtual evolution run.
3. *precision* How many sequences have to attributed correctly to a family or cluster in order to be proven to be correct. If you chose for instance 0.8 here a family of 1000 sequences can only be found in clusters that have between 800 and 1200 sequences and hold at least 800 sequences of the of the original partition.
4. *split-sets* The binary clusters layer files that are the result of an adaptive clustering run performed with the tools presented in section 4.12.

4.25.3 Example

```
simulation_verification sim.fasta 10 0.8 /tmp/s*
```

Listing 4.61 – An example of the *simulation_verification* tool.

In this example the simulation verification tool uses the simulated dataset sim.fasta which contains 10 partitions and investigates if clusters found during an adaptive clustering run yielded at least in one layer the correct attribution of a family under the precision criteria 0.8.

A possible output might be:

```
Found 8 out of 10 partitions
```

Listing 4.62 – Example output of the *simulation_verification* tool.

4.25.4 Implementation

The tool is implemented in *simulation_verification.c*.

4.26 pca2densitymap

4.26.1 General

The *pca2densiymap* tool allows you to visualize the the sequence density in a PCA file, as for instance created using the *fasta2kmer* tool outlined in section 4.1, by generating a portable network graphics (PNG) image. This basically produces a histogram by laying a rectangular grid across a projection into two/three dimensional subspace, spun by principal components. The tool then counts how many sequences are found within a square/cube and attributes this value to a pixel/voxel in the final image/images. The 3D version creates a stack of images. To overcome the limitations of an image based tool such as this one, one may refer to the *pca2densityfile* tool outlined in section 4.27.

4.26.2 Usage

The tool can be run using the following commands:

```
pca2densitymap [fasta] [pca] [dimensions] [2D or 3D] \
  [invertal-length] [png-file-or-files]
```

Listing 4.63 – Calling the *pca2densitymap* tool

with the arguments:

1. *fasta* The original FASTA file that has been transformed into a k-mer representation which was projected into a subspace spun by principal components. For instance by using the tools *fasta2kmer* and *kmer2pca* as described in sections 4.1 and 4.2.
2. *pca* A file containing the projections onto the principal components.
3. *dimensions* An integer describing how many dimensions the PCA subspace consists of.
4. *2D or 3D* One may choose 2 for a 2-dimensional output or 3 for a 3-dimensional output. 3-dimensional outputs are not fully tested though.
5. *interval-length* The grid spacing for our histogram.
6. *png-file-or-files* A path for the resulting PNG file. In case of a 3D output a whole stack of images with individual numbers for each slice will be created.

4.26.3 Example

```
pca2densitymap test.fasta test.pca 7 2 0.1 /temp/out.png
```

Listing 4.64 – Example of the *pca2densitymap* tool

In this example an image, *out.png* will be created that contains pixels ranging from white, representing the highest number of sequences found in a grid tile, to black, with no sequences in the boundaries of a grid tile. As the image can only encode 8 bits per pixel only 255 different gray levels are available. This might cause, due to the rescaling, that only a few white points exist in the image if the dataset contains very large peaks. *test.fasta* is the sequence dataset, *test.pca* its projection into a 7 dimensional PCA subspace. The tool always chooses to only treat either the first two or three dimensions, for the 2D or 3D case respectively. Further in this example, we ask for a two dimensional output with a grid spacing of 0.1 [k-mer frequencies]. The tool yields some information about the image generated:

```
   shift x:  8.45373440
   shift y:  5.56738997
n points x:    122
n points y:     89
```

```
factor 255/maximum(intensity): 0.10814250
```

Listing 4.65 – Text output of the *pca2densitymap* tool

where shift x and y are origin of the image, in general on the upper left. n Points x and y are the number of pixels according to the spacing chosen that are calculated and stored in the output PNG file. The last value is the scalefactor that has been used in order to scale the densities from 0 to 255. A sample of an image generated using this tool is highlighted in figure 4.7.

4.26.4 Implementation

The interface to this tool is implemented in *pca2densitymap.c*. The engine of the tool resides in *density.c*.

4.27 pca2densityfile

4.27.1 General

The *pca2densityfile* functions similar to the *pca2densitymap* tool outlined in section 4.26. It generates a two dimensional histogram across a 2 dimensional PCA projection by selection of a grid spacing. The tool then writes the densities for each tile to a text file that can be used for further processing, or external visualization tools.

4.27.2 Usage

The *pca2densityfile* tool can be called like:

```
pca2densityfile [fasta] [pca] [dimensions] [interval-length] > [densityfile]
```

Listing 4.66 – Calling the *pca2densityfile* tool

with the arguments:

1. *fasta* A FASTA file that has been transformed to a k-mer representation (i.e. using *fasta2kmer* as shown in section 4.1) and where such a k-mer representation has been projected onto a PCA subspace (i.e. using *kmer2pca* as outlined in section 4.2.)
2. *pca* The projections onto the principal components.

Figure 4.7 – The sequence density in a two dimensional PCA subspace of a k-mer representation. The image was upscaled and color inverted for printing using the Gnu Image Manipulation Program (GIMP) [13].

3. *dimensions* How many principal components have been retained, the dimensionality of the PCA subspace stored in the *pca* file.
4. *interval-length* The grid distance, i.e. the tile size of the histogram.
5. *densityfile* A text file where for each tile a density value is stored.

4.27.3 Example

```
pca2densitymap test.fasta test.pca 7 0.1 > /tmp/out
```

Listing 4.67 – Example of the *pca2densityfile* tool

In this example a dataset of sequences *test.fasta* and a representation in a 7 dimensional PCA subspace obtained from a k-mer representation of the sequences *test.pca* is examined. A histogram is generated using a grid spacing (tile size) of 0.1 [k-mer frequencies]. The number of sequences per tile is printed to the output file */tmp/out*.

```
   shift x:  8.45373440
   shift y:  5.56738997
n points x:    122
n points y:     89
0 0 0
1 0 0
2 0 0
3 0 0
4 0 0
5 0 0
6 0 0
7 0 0
8 0 0
9 0 0
10 0 0
11 0 0
12 0 0
13 0 0
14 0 0
15 0 0
```

Listing 4.68 – Example output of the *pca2densityfile* tool

In the beginning of the output file a shift x and y value which tells us about the origin of the file is found. Further the file shows us how many tiles were generated according to the *interval-length* parameter. This header is followed by the data which is represented as three values per line. Following the header the data is represented as follows: On each line the x and y coordinate (of

the tile) i.e. the number of tile that you are in, and the number of sequences present in this square are shown. Such a file can be used with 3rd party visualization tools, such as Blender [14] in order to inspect the density of sequences in three dimensions. An example is shown in figure 4.8. We also provide a Blender import script for Blender 2.80 and 2.90 in the following:

```
import bpy

def add_map_from_density_file(filename, objname, delta, scalefactor):
    f = open(filename, "r")

    line = f.readline();
    line_array = line.split();
    shift_x = float(line_array[2]);

    line = f.readline();
    line_array = line.split();
    shift_y = float(line_array[2]);

    line = f.readline();
    line_array = line.split();
    n_points_x = int(line_array[3]);

    line = f.readline();
    line_array = line.split();
    n_points_y = int(line_array[3]);

    verticies = [];
    faces = [];

    n_points = n_points_x*n_points_y

    for i in range(n_points):
        line = f.readline()
        line_array = line.split();
        current_point = [(delta*float(line_array[0])-shift_x,
                          delta*float(line_array[1])-shift_y,
                          float(line_array[2])*scalefactor)]
        verticies.extend(current_point)

    for j in range(n_points_y-1):
        for i in range(n_points_x-1):
            current_face = [(j*n_points_x+i,(j+1)*n_points_x+i,
                             (j+1)*n_points_x+i+1,j*n_points_x+i+1)]
            faces.extend(current_face)

    themesh = bpy.data.meshes.new(objname)
    theobject = bpy.data.objects.new(objname, themesh)

    bpy.data.objects.new(objname, themesh)
    bpy.context.scene.collection.objects.link(theobject)

    themesh.from_pydata(verticies, [], faces)
    themesh.update(calc_edges=True)

    return theobject

rescale = 5.;
```

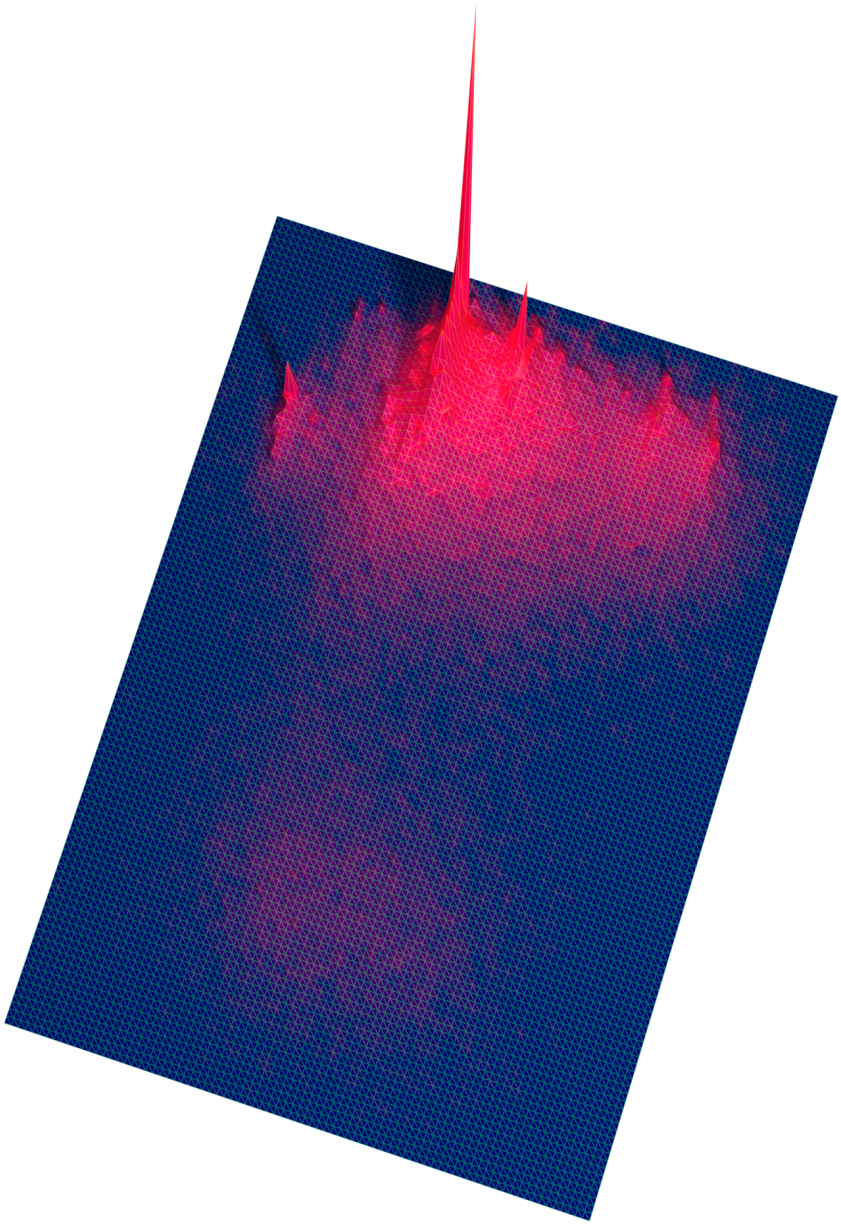


Figure 4.8 – The output of *pca2densityfile* visualized with Blender [14]

```
add_map_from_density_file("/tmp/desityfile", "surface-name", 0.1, 0.03/rescale)
```

Listing 4.69 – Python script to import the output of the *pca2densityfile* tool into Blender.

where you have to adapt */tmp/densityfile* to the location of your output, *surface-name* to the name that you want to attribute to your 3D surface, *0.1* to the grid spacing that you have chosen and *0.03* to your rescale factor.

4.27.4 Implementation

The interface to this tool is implemented in *pca2denistyfile.c*. The inner workings are implemented in *density.c*.

4.28 reverse_with_mask

4.28.1 General

This tool allows you to reverse selected sequences in a FASTA file. A mask text file has to be provided to indicate which sequences shall be reversed. This tool only changes the read direction of the of the selected sequences. A more elaborate tool is the *reverse_complement_with_mask* tool found in section 4.29.

4.28.2 Usage

The *reverse_with_mask* tool can be called like:

```
reverse_with_mask [fasta] [mask] > [fasta-out]
```

Listing 4.70 – Calling the *reverse_with_mask_tool* tool

with the following arguments:

1. *fasta* A FASTA file to reverse sequences in.
2. *mask* A mask file to indicate the utility which sequences to reverse. The file has to contain a line with a number for each sequence. If the number matches 1 the sequence corresponding to this line in the mask file is reversed in the FASTA output file.
3. *fasta-out* The FASTA dataset of all sequences with those reversed where the mask matches.

4.28.3 Example

Imagine the following FASTA file with three sequences:

```
> sequence_1
ACTG
> sequence_2
ACCT
> sequence_3
GGAT
```

Listing 4.71 – Example of the *reverse_with_mask* tool - FASTA file

As we would like to reverse the second sequence we create the following mask file:

```
0
1
0
```

Listing 4.72 – Example of the *reverse_with_mask* tool - Mask file

Calling the tool as outlined below:

```
reverse_with_mask test.fasta test.mask > out.fasta
```

Listing 4.73 – Example of the *reverse_with_mask* tool

will produce the following FASTA file output:

```
> sequence_1
ACTG
> sequence_2
TCCA
> sequence_3
GGAT
```

Listing 4.74 – Example of the *reverse_with_mask* tool - Output FASTA file

4.28.4 implementation

The tools interface is implemented in *reverse_with_mask.c*. The code that reverses the sequences resides in *dataset.c*.

4.29 reverse_complement_with_mask

4.29.1 General

This tool allows you to reverse complement selected sequences in a FASTA file. A mask text file has to be provided to indicate which sequences shall be reversed. This tool not only changes read direction of the selected sequences but also correctly complements the nucleotides.

4.29.2 Usage

The *reverse_complement_with_mask* tool can be called like:

```
reverse_with_mask [fasta] [mask] > [fasta-out]
```

Listing 4.75 – Calling the *reverse_complement_with_mask_tool* tool

with the following arguments:

1. *fasta* A FASTA file to reverse complement sequences in.
2. *mask* A mask file indicating the utility which sequences to reverse complement. The file has to contain a line with a number for each sequence. If the number matches 1 on line of the mask file, the sequence corresponding to this line is replaced by its reverse complement in the resulting FASTA file.
3. *fasta-out* The FASTA dataset containing all sequences together with the reverse complemented at location where the mask matched.

4.29.3 Example

We imagine the following FASTA file with three sequences:

```
> sequence_1
ACTG
> sequence_2
ACCT
> sequence_3
GGAT
```

Listing 4.76 – Example of the *reverse_complement_with_mask* tool - FASTA file

As we would like to reverse complement the second sequence, we write a mask file with the following contents:

```
0
1
0
```

Listing 4.77 – Example of the *reverse_complement_with_mask* tool - Mask file

Running the tool by issuing the following command:

```
reverse_complement_with_mask test.fasta test.mask > out.fasta
```

Listing 4.78 – Example of the *reverse_complement_with_mask* tool

produces the following *out.fasta* file:

```
> sequence_1
ACTG
> sequence_2
AGGT
> sequence_3
GGAT
```

Listing 4.79 – Example of the *reverse_complement_with_mask* tool - Output FASTA file

4.29.4 Implementation

The tools interface is implemented in *reverse_complement_with_mask.c*. The code that reverses the sequences resides in *dataset.c*.

4.30 digest_X

The *digest_X* tools allow you to virtually digest a sequence using restriction enzymes creating a FASTA file with the sequences cut into pieces at the corresponding restrictions sites. Currently the following tools corresponding to the following restriction enzymes are implemented: *digest_XbaI, digest_XmnI* and *digest_HindIII*.

4.30.1 Usage

The tools can be used in the following manner:

```
digest_XbaiI [fasta] [n-threads] > [fasta-out]
digest_XmnI [fasta] [n-threads] > [fasta-out]
digest_HindIII [fasta] [n-threads] > [fasta-out]
```

Listing 4.80 – Calling the *digest_X* tools

with the arguments:

1. *fasta* A FASTA file. Only the first sequence of the *fasta* file will be digested.
2. *n-threads* This parameter indicates how many threads are available to the digestion process.
3. *fasta-out* A fasta file containing the digested sequences.

4.30.2 Example

```
digest_HindIII test.fasta 8 > digested.fasta
```

Listing 4.81 – Example of the *digest_X* tools

The command above digests the first sequence stored in *test.fasta* and stores the resulting sequences in *digested.fasta*. The digestion process can use up to eight threads on the machine it runs on. The restriction enzyme used for this virtual digestion is HindIII.

4.30.3 Implementation

The interface to the *digest_X* tools is implemented in *digester.c* The digestion engine is implemented in *restriction_digest.c*.

4.31 replace_N_sequence

4.31.1 General

replace_N_sequence Is a tool to replace sequences that are holding N letters in them with all possible combinations possible. This tool currently only works with nucleotide sequences.

4.31.2 Usage

The tool can be called like:

```
replace_N_sequence [fasta] [sequence-index] [fasta-out]
```

Listing 4.82 – Calling the *replace_N_sequence* tool

where the arguments are as follows:

1. *fasta* A FASTA file holding sequences
2. *sequence-index* An index to select a sequence from the *fasta* file. In this tool the first sequence is selected by 1 (and not 0).
3. *fasta-out* All possible permutations by replacing the N letters in the sequence.

4.31.3 Example

Let us imagine a FASTA file like the following:

```
>sequence1
ACGT
>sequence2
ANNT
>sequence3
ANTG
```

Listing 4.83 – A FASTA example for the *replace_N_sequence* tool.

We would like to obtain all sequences possible by replacing the N letters in the second sequence and hence, we call the utility:

```
replace_N_sequence test.fasta 2 /tmp/out.fasta
```

Listing 4.84 – Fasta example for the *find_satellite* tool

which yields a */tmp/out.fasta* file with the following contents:

```
>sequence_0
AAAT
>sequence_1
ACAT
>sequence_2
AGAT
>sequence_3
ATAT
>sequence_4
AACT
>sequence_5
ACCT
>sequence_6
AGCT
>sequence_7
ATCT
>sequence_8
AAGT
>sequence_9
ACGT
>sequence_10
AGGT
>sequence_11
ATGT
>sequence_12
AATT
>sequence_13
ACTT
>sequence_14
AGTT
>sequence_15
ATTT
```

Listing 4.85 – A FASTA output example for the *replace_N_sequence* tool.

4.31.4 Implementation

The tools interface is implemented in *replace_N_sequence.c*. The mechanism is implemented in *kmers.c*.

4.32 tree_pureness

4.32.1 General

tree_pureness is a tool that can compare a tree and the layer wise cluster partitions generated by an adaptive clustering run, for instance, with the tools outlined in section 4.12 with a ground trouth set. Ground trouth sets can for instance be created using the
split_set_from_annotation (c.f. section 4.17) or
split_set_from_swarm (c.f. 4.17) tool if a comparison against such tools is the reason for using this utility. *Tree_purness* calculates the following tree indices:

1. Pureness Index,
2. Cluster correspondance, and
3. Impure over pure.

This tool is the major tree quality measurement tool in MNHN-Tree-Tools if the ground truth for a tree is known.

4.32.2 Usage

The tool can be called like

```
tree_pureness [fasta] [ground-truth] [split-sets] > [output]
```

Listing 4.86 – Calling the *tree_pureness* tool

with the following arguments:

1. *fasta* A FASTA dataset of sequences that was used to build a tree from, and that has a known a ground truth set of partitions.
2. *ground-truth* A binary *split-set* file that contains the partitions of the ground truth.
3. *split-sets* The resulting *split-set*s from an adaptive clustering run, that represent a full tree. (Such a set can be obtained by the tools outlined in section 4.12.)

4. *output* The output is a text file containing for each layer in a tree all three indices, as outlined above, that enable an efficient comparison to a ground truth partition.

4.32.3 Algorithm

The program defines the three indices:

1. *Pureness:* We define the Pureness Index to be:

$$P_l = \frac{1}{n_{l,\text{inp}}} \sum_{i=1}^{n_{l,\text{inp}}} \sum_{k=1}^{n_{\text{target}}} \frac{f_{l,i,k}^{<50\%}}{C_{l,i}}, \tag{4.23}$$

   where l represents the layer of a tree, $n_{l,\text{inp}}$ the number of impure clusters and hence, the clusters that are mixture of two partitions from the initial truth dataset to compare to. n_{target} is the number of partitions in the original truth dataset, $f_{l,i,k}^{<50\%}$ the number of elements belonging to partition k of the truth dataset in the impure cluster indexed by i in layer l if this number represents less then 50% of the elements in the inpure cluster otherwise $f_{l,i,k}^{<50\%}$ represents the total number of elements in the impure cluster minus the elements in this cluster that belong to the partition of the truth dataset that is indexed by k. Finally $C_{l,i}$ represents the total number of elements in the impure cluster in layer l indexed by i.

2. *Cluster correspondance*: It is straight forward to reason that pureness as define above is not enough, as a partitioning with enough small clusters would yield highly pure clusters but not a meaningful partitioning, as such we further define:

$$S_l = \frac{|n_l - n_{\text{target}}|}{n_{\text{target}}}, \tag{4.24}$$

   where n_l is the number of clusters in layer l to be the measure of correspondance between the number of clusters in the truth partioning and the number of clusters obtained in our partitioning.

3. *Impureness over pureness*: Finally we argue that we require a third value in order to describe the quality of our tree:

$$I_l = \frac{n_{l,\text{inp}}}{n_l}. \tag{4.25}$$

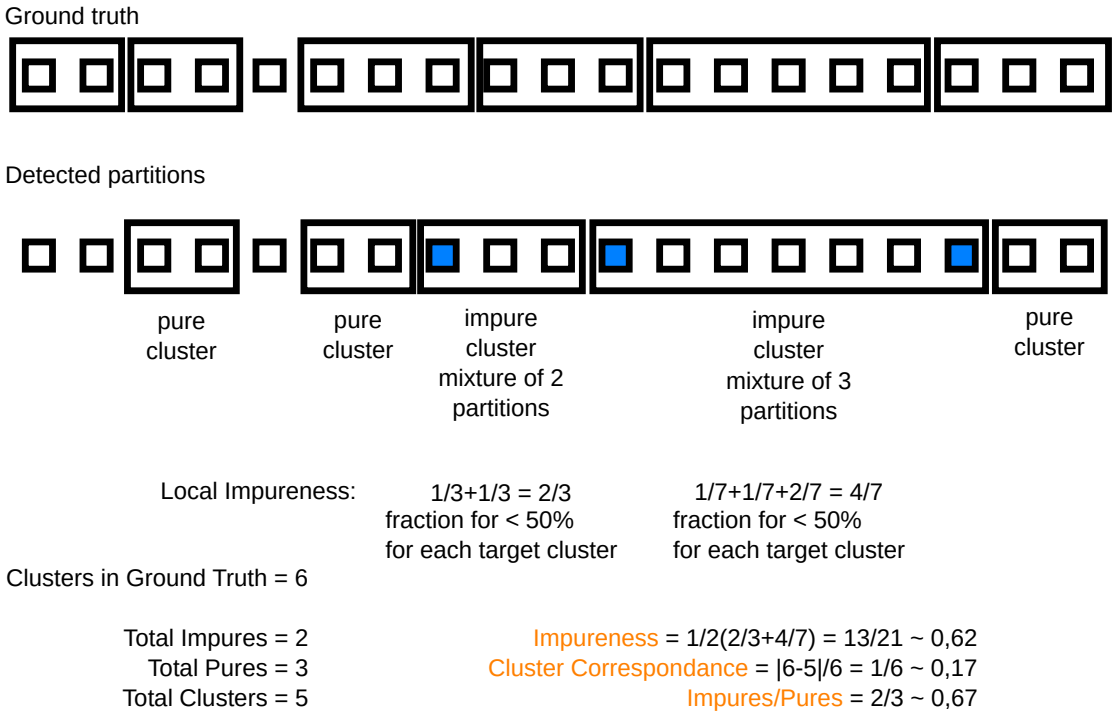


Figure 4.9 – A ground truth and a computed clustering. The figure shows how the cluster indices: Pureness, Cluster Correspondence and Impureness over Pureness are calculated.

> I_l represents the number of impure clusters, hence those who contain mixtures of partitions from the truth partitioning over the number of clusters that are pure.

An example of clustering and the calculation of the three indices is shown in figure 4.9.

4.32.4 Example

We call *tree_purness* like

```
tree_purness test.fasta ground-truth /tmp/s* > /tmp/pureness
```

Listing 4.87 – Example of the *tree_pureness* tool

where *test.fasta* is a dataset and *ground-truth* is a binary *split-set* describing the partition we are comparing all the clusterings in the tree to. */tmp/s** is the wildcard to select all the binary

clusterings that have been produced for a complete tree using an adaptive clustering run (c.f. section 4.12). A resulting */tmp/pureness* file might look like the following:

```
0.207187        1.015385        0.259542
0.207658        0.761538        0.288210
0.214303        0.407692        0.267760
0.267122        0.230769        0.281250
0.270319        0.007692        0.297710
0.278394        0.176923        0.280374
0.333774        0.330769        0.264368
0.301733        0.415385        0.315789
0.327885        0.569231        0.267857
```

Listing 4.88 – Example output of the *tree_pureness* tool

The output presents three tabulator separated values per line, where the first value is the Pureness, the second the Cluster Correspondence and the third the Impureness over Pureness value. Using a tool like *gnuplot* for instance one can create a three value diagram in order to judge if a level of the tree yields the accuracy requested to the ground truth. Such an example is shown in figure 4.10.

4.32.5 Implementation

The interface is implemented in *tree_purness.c*. The functionality is to be found in *cluster_io.c*.

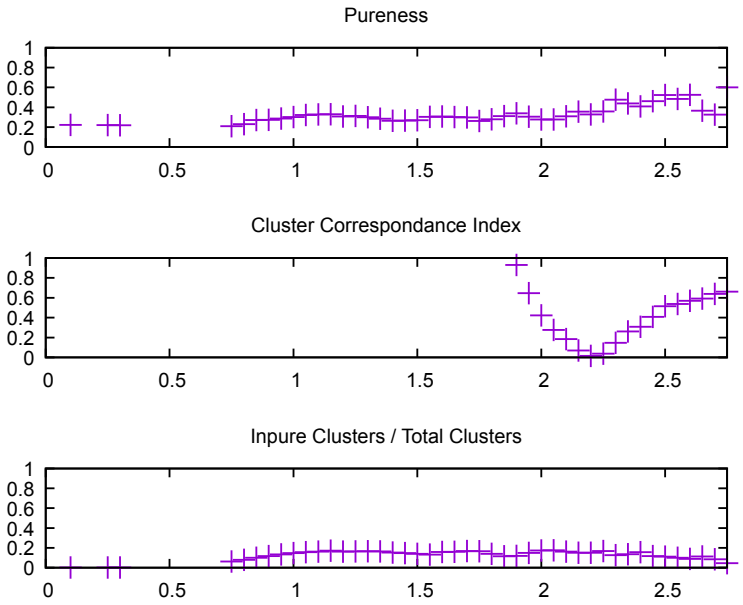


Figure 4.10 – An example of a visual representation of the data obtained from the *tree_pureness* tool. On the abscissa the ϵ values shown reside in the output of the adaptive clustering run (c.f. section 4.12).

A | Bibliography

[1] Lipman D, Pearson W. Rapid and sensitive protein similarity searches. Science. 1985;227(4693):1435–1441. Available from: https://science.sciencemag.org/content/227/4693/1435.

[2] Ester M, Kriegel HP, Sander J, Xu X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. KDD'96. AAAI Press; 1996. p. 226–231.

[3] Rousseeuw PJ. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics. 1987;20:53 – 65. Available from: http://www.sciencedirect.com/science/article/pii/0377042787901257.

[4] Rice P, Longden I, Bleasby A. EMBOSS: The European Molecular Biology Open Software Suite. Trends in Genetics. 2000 Jun;16(6):276–277. Available from: https://doi.org/10.1016/S0168-9525(00)02024-2.

[5] Williams T, Kelley C. Gnuplot 5.2: an interactive plotting program; 2017. http://gnuplot.info/. Available from: http://gnuplot.info.

[6] Cacheux L, Ponger L, Gerbault-Seureau M, Richard FA, Escudé C. Diversity and distribution of alpha satellite DNA in the genome of an Old World monkey: Cercopithecus solatus. BMC Genomics. 2016 Nov;17(1):916. Available from: https://doi.org/10.1186/s12864-016-3246-5.

[7] Olsen G. Interpretation of the "Newick's 8:45" Tree Format Standard; 1990. https://evolution.genetics.washington.edu/phylip/newick_doc.html.

[8] Junier T, Zdobnov EM. The Newick utilities: high-throughput phylogenetic tree processing in the Unix shell. Bioinformatics. 2010 05;26(13):1669–1670. Available from: https://doi.org/10.1093/bioinformatics/btq243.

[9] GNU P. Free Software Foundation. Bash (3.2. 48)[Unix shell program]; 2007.

[10] Kahan W. Pracniques: Further Remarks on Reducing Truncation Errors. Commun ACM. 1965 Jan;8(1):40. Available from: https://doi.org/10.1145/363707.363723.

[11] Anderson E, Bai Z, Dongarra J, Greenbaum A, McKenney A, Du Croz J, et al. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing. Supercomputing '90. Washington, DC, USA: IEEE Computer Society Press; 1990. p. 2–11.

[12] Mahé F, Rognes T, Quince C, de Vargas C, Dunthorn M. Swarm v2: highly-scalable and high-resolution amplicon clustering. PeerJ. 2015;3:e1420.

[13] The GIMP Development Team. GIMP;. Available from: https://www.gimp.org.

[14] Community BO. Blender - a 3D modelling and rendering package. Stichting Blender Foundation, Amsterdam; 2020. Available from: http://www.blender.org.