# PhyloMeasures: A Package for Computing Phylogenetic Biodiversity Measures and Their Statistical Moments

Constantinos Tsirogiannis
Department of Bioscience and MADALGO*
Aarhus University, Denmark
constant@madalgo.au.dk

Brody Sandel
Department of Bioscience and MADALGO*
Aarhus University, Denmark
brody.sandel@bios.au.dk

*Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

# Contents

# 1 Introduction

Researchers in biology are frequently confronted with the following problem: given a set of species, measure whether these species are close evolutionary relatives. The most common way to measure this is to use a phylogenetic tree $\mathcal{T}$, where each leaf node corresponds to a species, and the tree edges are assigned a distance value representing e.g. time since the last speciation event. From $\mathcal{T}$ a sample (subset) of leaves $S$ are selected, corresponding to the species that we want to examine. The next step is to choose a measure that expresses the distance between the leaves in $S$ based on the structure of $\mathcal{T}$. In the related literature, such methods are refered to as *phylogenetic biodiversity measures*. There exist several methods of this kind, and among the most popular ones are *Phylogenetic Diversity* (PD) [4], *Mean Pairwise Distance* [17] and the *Mean Nearest Taxon Distance* (MNTD) [18]. Given a tree $\mathcal{T}$ and a sample of its leaves $S$, each of these measures returns a value that represents the distance between the leaves in $S$ (we discuss later how each measure computes this value). A larger distance value indicates that the species represented in $S$ are distant evolutionary relatives, while a smaller value shows that they are closely related.

However, deciding if a value calculated by a measure $M$ is relatively large or small is not a straightforward task. To do this, we need to compare this value with the values that $M$ returns for other samples of species in $\mathcal{T}$. More specifically, we would like to do this comparison with the values that $M$ returns for samples of leaves that have exactly the same number of elements as $S$. Let $s = |S|$ denote the number of leaves in $S$, and let $M(\mathcal{T}, S)$ indicate the value of measure $M$ for the sample $S$ of $\mathcal{T}$. Ideally, for a given tree $\mathcal{T}$ we would like to compare $M(S)$ with the values of $M$ for all possible samples of leaves in $\mathcal{T}$ that consist of exactly $s$ leaves. The standard way to do this comparison is to calculate a value that is based on the expectation and standard deviation of $M$ over all possible sets of leaves of size $s$ [18]. For a measure $M$, this value is equal to:

$$\frac{M(\mathcal{T}, S) - \mu_M(\mathcal{T}, s)}{\sigma_M(\mathcal{T}, s)},$$

where $\mu_M(\mathcal{T}, s)$ and $\sigma_M(\mathcal{T}, s)$ are respectively the expectation (mean) and the standard deviation of the measure $M(\mathcal{T}, S)$ among all samples of leaves in $\mathcal{T}$ with $s = |S|$ elements. A common way to calculate the expectation and deviation values is to consider that every sample of exactly $s$ species in $\mathcal{T}$ is picked uniformly out of all possible samples of the same size. We call this value the *standardised value* of measure $M$. To avoid any confusion, we call $M(\mathcal{T}, S)$ (that is the non-standardised value of the measure) the *basic* value of $M$ for sample $S$.

Therefore, experts in phylogenetic research are in need of a software tool that can compute not only the basic values of standard biodiversity measures, but also their expectation and deviation. Such a tool must be very efficient; the size of available phylogenetic tree datasets can reach up to hundreds of thousands of leaves, and a typical case study may involve calculating the standardised value for thousands of leaf samples in such a tree. Hence, a program that computes the standardised value of a phylogenetic measure must run very fast even for large trees. Otherwise, processing datasets of this size becomes infeasible. Today, there exists software for computing the basic values of phylogenetic measures for specific samples of species; for example, packages `picante` and `phylodiv` [8, 10, 11] which are available through the R software platform. However, except for the PD, these packages do not provide functions for computing exactly the expectation and the deviation of phylogenetic measures.

Due to the absence of other software tools, researchers are forced to calculate the statistical moments of most phylogenetic measures using the following inexact technique; for a given sample size $s$, a few samples of exactly $s$ leaves are selected at random from the examined tree $\mathcal{T}$. Then, the expectation and the deviation of a measure $M$ is calculated using the basic values of $M$ only for the

selected samples. The number of the selected samples is usually around a thousand. When $\mathcal{T}$ and $s$ are sufficiently large, a thousand samples is a very small amount compared to the number of all possible samples of this size. This implies that the sampling approach is inexact, and may yield estimated values for the expectation and the deviation that are very different from the original ones. Also, depending on the number of samples used, this approach can be very slow. Therefore, there is the need for a software package that, on one hand, can compute the statistical moments of popular phylogenetic measures *exactly*, and also can process large tree datasets *efficiently*.

Inspired by the above we introduce `PhyloMeasures`; a software package that provides efficient programs for calculating phylogenetic biodiversity measures and their statistical moments. In particular, we provide efficient programs for nine different measures. Four of these are measures that calculate a diversity value for a single sample of species. These are are the (unrooted) PD, MPD, MNTD [4, 17], and the *Core Ancestor Cost* (CAC) that we introduced in a previous paper [15]. The remaining five measures are measures of $\beta$-diversity; that is a measure that calculates a diversity value between two sample of species. These are the the *Common Branch Length* (CBL), *Community Distance* (CD), the *Community Distance Nearest Taxon* (CDNT) [6, 13], the *Phylogenetic Sorensen's Similarity* (PhyloSor) [6, 13], and the *Unique Fraction* (UniFrac) [9]. The $\beta$-diversity measures are also known as *two-sample* measures, as opposed to the rest of the phylogenetic measures which are known as *single-sample* measures. For each of the nine measures that we mentioned, `PhyloMeasures` provides a function that computes the basic value of the measure for specific input samples. Also, for most of these measures (except CDNT, PhyloSor and UniFrac) the package provides efficient functions that compute the expectation and the deviation among all possible leaf samples of a given size, and the standardised value of the measure for given input samples.

`PhyloMeasures` is open-source software (published under the GNU Public License, version 3), and is available both as an R package and as a C++ library. `PhyloMeasures` has two main advantages; one, it is the first software tool that provides programs for computing exactly the expectation and the deviation for several phylogenetic biodiversity measures. Two, all functions provided in the package are designed to run very fast even for large tree datasets. To do this, we developed our programs based on the performance paradigm that is used in the field of Algorithms Design. According to this paradigm, we designed algorithms whose performance scales nicely as the input tree size increases. As a result, for most functions that we implemented the running time is comparable to the time needed just for reading the input tree $\mathcal{T}$. We also conducted experiments that measure the efficiency of the `PhyloMeasures` functions in practice. The results of these experiments show how the running time of our functions scales for different input tree sizes.

More than that, the C++ version of `PhyloMeasures` package provides the following feature; the user can tune the functions of the package so that they perform exact numerical operations. This can be used to eliminate any precision issues that appear when computing the statistical moments of certain measures. As we show later, even without using this option the `PhyloMeasures` package produces results of high precision when processing large tree datasets.

The rest of this document is organised as follows; In the next section we provide an extensive overview of the package. There we outline the functions that are available in the package, and we present theoretical bounds for their performance. We also show experimental results that prove the efficiency of our programs in practice. In Section 3 we present in detail the interface of every function that is available in the R version of the package. Then, in Section 4 we describe the structure of the C++ package and the interface of the functions that are provided there. In Section 5 we discuss issues of numerical precision that arise when computing the moments of certain phylogenetic measures. We describe experiments on trees that consist of a few thousand leaves, where we show that the output

returned by our programs has very high precision. We also show how the C++ package can be tuned to use custom multiprecision number types, to achieve computations of arbitrary precision.

# 2 Package Overview

## 2.1 The Functions Provided in the Package

The `PhyloMeasures` package offers functions for calculating nine different phylogenetic measures and their statistical moments. Four of these measures are single-sample measures (Phylogenetic Diversity, Mean Pairwise Distance, Mean Nearest Taxon Distance, and Core Ancestor Cost) and the remaining five are two-sample measures (Common Branch Length, Community Distance, Community Distance Nearest Taxon, Phylogenetic Sorensen's Similarity, and Unique Fraction). In Section 6 we provide a formal definition for each of these measures. For most of the measures (that is except CDNT, PhyloSor, UniFrac) the package provides three functions; a function for evaluating the basic or the standardised value of the measure given a specific sample (or a specific pair of samples for $\beta$-diversity measures), a function for computing the expectation of the measure among all leaf samples of a given size, and a function for computing the deviation of the measure. We call the first of these functions a *query* function, and we refer to the other two functions as *moment* functions. In most applications, it is required to compute the value of a phylogenetic measure for many different samples of leaves. For this reason, each query function in `PhyloMeasures` is developed so as to process several queries in a single call.

For the CAC, apart from the three functions described above, the package includes one extra moment function; this function computes the first $k$ statistical moments of this measure for any given integer $k > 0$. For CDNT, PhyloSor, and UniFrac, the package does not provide any moments functions. However, there are more than one query functions available for the CDNT. This is because, unlike the other two-sample measures, the CDNT is not symmetric (see Section 6 for the definition of this measure). That means, for a given tree $\mathcal{T}$ and two samples of leaves $A$ and $B$ the CDNT *from* sample $A$ *to* sample $B$ is not necessarily equal to the CDNT from $B$ to $A$. More details on the functions available for the CDNT can be found in Sections 3.3 and 4.4.7.

**Structure of the Different Package Versions** `PhyloMeasures` is available both as an R package and as a C++ library. The interface and the structure of `PhyloMeasures` differs between these two package versions. The interface of the R package has a simple structure; it is a set of separate functions, the query and moment functions that we described above. The C++ package follows the object-oriented programming paradigm, and is structured as a hierarchy of different classes. The R package is built on top of the C++ package, and is fundamentally an R interface for the main C++ functions. The C++ package offers a few more functionalities; there are available functions for handling phylogenetic trees (for example, to check if a tree is ultrametric). Also, the C++ package provides the opportunity to select which number type is used during the execution of the query and moment functions. Therefore, the user can provide a number type that is more precise than the standard built-in C++ types `float` and `double`. This can reduce the precision errors that appear when using floating point arithmetic. For more information on this issue, see Sections 4.1 and 5.

Next we present one of the key advantages of `PhyloMeasures`; we describe how efficient the package functions are. For every function in `PhyloMeasures` we provide a measure for its performance, and we show how the running time of the function is expected to grow in practice with respect to the input size.

## 2.2 The Efficiency of the Package Functions

When designing an algorithm, we want to make it as efficient as possible, especially if we need to process large amounts of data. There are many ways to measure the efficiency of an algorithm; we could implement this algorithm and then measure its running time in practice. But we can also measure the efficiency of an algorithm in a different way, without having to implement the algorithm itself. Instead of measuring the actual running time, we can analyse an algorithm "on paper" and count the number of basic operations that would take place during its execution. When refering to basic operations we mean basic numerical operations (such as additions, multiplications, comparisons, etcetera), but also other simple operations that take place during the execution of a program. These can be, for instance, creating a new variable, or assigning a value to a variable. Ideally, we would like to find how the number of these operations scales when we execute the algorithm on inputs of different sizes.

In particular, let $A$ be an algorithm, and suppose that we have an input dataset for this algorithm that consists of $n$ elements; for example a phylogenetic tree $\mathcal{T}$ that has $n$ nodes. In the field of Algorithms Design, the standard way to measure the performance of $A$ is to find an upper bound on the number of operations that $A$ performs given any input of size $n$. More specifically, this bound is expressed as a mathematical function of $n$; we want to prove that the $A$ takes at most $f(n)$ operations to process any input of $n$ elements, where $f(\cdot)$ is an increasing function with $f(x) > 0$ for any $x > 0$. By selecting $f(n)$ appropriately, we get a good estimation of how the performance of the algorithm scales as $n$ increases.

In fact, it is not important to pick $f(n)$ precisely; for example, suppose $A$ takes at most $g(n) = 5n^2 + 100n + \log n$ operations when processing any input of size $n$. In practice, we are only interested in the part of $g(n)$ that grows fastest as $n$ increases, that is the $n^2$ term in the function. It is this term that defines the performance of $A$ for large values of $n$. To express this in short, we say that $A$ takes $O(n^2)$ operations for any input of size $n$. The notation $O(f(n))$ is used to indicate that the number of operations in $A$ grows roughly as fast as $f(n)$ when $n$ becomes large. For an algorithm $A$ that takes $O(f(n))$ operations to process any possible input of size $n$, we say that the *worst case running time* of $A$, or the *worst case time complexity* of this algorithm is $O(f(n))$. The $O(\cdot)$ notation is the so-called *big O* ("big oh") notation. For a more formal definition of this notation, the reader may refer to a standard textbook on Algorithms Design [3].

We should note that, even if an algorithm has a small worst case time complexity in theory, this does not always guarantee that it will be efficient in practice. For instance, the most popular algorithm for sorting a set of $n$ numbers has a time complexity $O(n^2)$ at worst case, but performs better in practice than algorithms that have $O(n \log n)$ worst case complexity. However, it is widely accepted that algorithms with small worst case time complexity usually lead to more efficient programs in practice.

In a series of research papers, we showed how to design efficient algorithms for evaluating phylogenetic measures and their moments [16, 14, 15]. These algorithms were designed so as to have a small worst case time complexity, as defined above. Based on these algorithms we developed the functions that are available in the `PhyloMeasures` package. Table 1 shows the time complexity for every query and moment function in the package when the input is a tree $\mathcal{T}$ of $n$ nodes.

We see that most functions in the package have time complexity $O(n)$; this is the best that we can expect for any program that processes a tree of this size. Obviously, at least $n$ operations are needed only for reading a tree of this size from the input. In that sense, the algorithms used for developing these functions have an optimal performance.

For some package functions, the term $\mathrm{SI}(\mathcal{T})$ appears in the definition of the worst case time

Table 1: The worst case time complexity of the functions provided in `PhyloMeasures`. Each row shows the time complexity of the functions that are available for each measure. Column "Basic Value" shows the time complexity of the functions that compute the basic value of a measure for a single sample of leaves (for single-sample measures), or for a pair of samples (for $\beta$-diversity measures). We use "-" to indicate that a function is not available for some measure. The functions that compute the expectation and the deviation of the MNTD work only for ultrametric trees (highlighted in the table). There are no functions available for computing the expectation and the deviation of CDNT, PhyloSor, and UniFrac.

| Measure | Basic Value | Expectation | Deviation | First $k$ Moments |
|---|---|---|---|---|
| Phylogenetic Diversity | $O(n)$ | $O(n)$ | $O(\mathrm{SI}(\mathcal{T}))$ | - |
| Mean Pairwise Distance | $O(n)$ | $O(n)$ | $O(n)$ | - |
| Mean Nearest Taxon Distance | $O(n)$ | $O(n)$ | $O(\mathrm{SI}(\mathcal{T}))$ | - |
| Core Ancestor Cost | $O(n)$ | $O(n)$ | $O(n)$ | $O(\mathrm{SI}(\mathcal{T}) + k)$ |
| Common Branch Length | $O(n)$ | $O(n)$ | $O(\mathrm{SI}(\mathcal{T}))$ | - |
| Community Distance | $O(n)$ | $O(n)$ | $O(n)$ | - |
| Community Distance Nearest Taxon | $O(n)$ | - | - | - |
| Phylogenetic Sorensen's Similarity | $O(n)$ | - | - | - |
| Unique Fraction | $O(n)$ | - | - | - |

complexity. Here, the term $\mathrm{SI}(\mathcal{T})$ is the so-called *Sackin's Index* of the tree $\mathcal{T}$ [2]. The Sackin's index is equal to the sum of the depths of all leaf nodes in $\mathcal{T}$. The depth of a leaf is defined as the number of tree edges that appear on the path between the leaf and the root node. The Sackin's index is mainly used in the literature to measure how balanced a phylogenetic tree is. Therefore, the performance of an algorithm that has time complexity $O(\mathrm{SI}(\mathcal{T}))$ depends on how balanced $\mathcal{T}$ is. For perfectly balanced trees the maximum depth among all leaves is $O(\log n)$. In that case, the value of Sackin's Index is $O(n \log n)$. However, for skewed trees the Sackin's Index may be much higher, up to $O(n^2)$. Fortunately, most available phylogenetic tree datasets are relatively balanced. Therefore, the package functions whose performance depends on the Sackin's Index of $\mathcal{T}$ are very efficient in practice.

Next, we present experiments that we conducted for measuring the efficiency of the `PhyloMeasures` functions on different kinds of phylogenetic data. There we show that the algorithms that we developed are not only supposed to be efficient in theory, but they also run very fast in practice.

### 2.3 Experiments

We measured the performance of the functions in `PhyloMeasures` by computing queries and moments of phylogenetic measures on trees of various sizes. All experiments that we describe in this section were executed using the R version of the package on a workstation with an Intel core i5-2430M CPU. This is a four-core processor with 2.40GHz per core. The main memory of this computer is 7.8 Gigabytes. Our implementations run on a Linux Ubuntu operating system, release 12.04. We executed the experiments using R version 3.1.2 (Pumpkin Helmet).

We performed two sets of experiments; in the first set of experiments we used phylogenetic trees that were extracted from real-world biological data. The dataset that we used is a phylogenetic tree representing the phylogeny of all mammal species [1]. This tree has 4,510 leaves and 6,618 nodes in total. We refer to this tree as the `mammals` dataset. From this tree we extracted eighteen

subtrees; each subtree has $250k + 10$ leaves with $k$ ranging from one to eighteen. The subtrees were produced by successively pruning chunks of 250 leaves from the original tree of 4,510 leaves. We denote the set of these trees by $\mathcal{U}$. For each tree $\mathcal{T} \in \mathcal{U}$ we produced a set $sm(\mathcal{T})$ of twenty samples of species, each sample having a different size. These samples were produced by selecting leaves in $\mathcal{T}$ uniformly at random. For each measure that is supported in `PhyloMeasures`, we measured the total time that it takes for the corresponding function in the package to compute a) the basic, and b) the standardised value of this measure for all samples in $sm(\mathcal{T})$. Also, for every moment function available in `PhyloMeasures` and for each tree $\mathcal{T} \in \mathcal{U}$, we measured the total time taken by the function to compute both the expectation and the deviation of the measure on $\mathcal{T}$ *for all* sample sizes in the range $\{1, 2, \ldots, \lfloor n/2 \rfloor\}$. The results of these experiments are illustrated in Figures 1, 2, and 3.

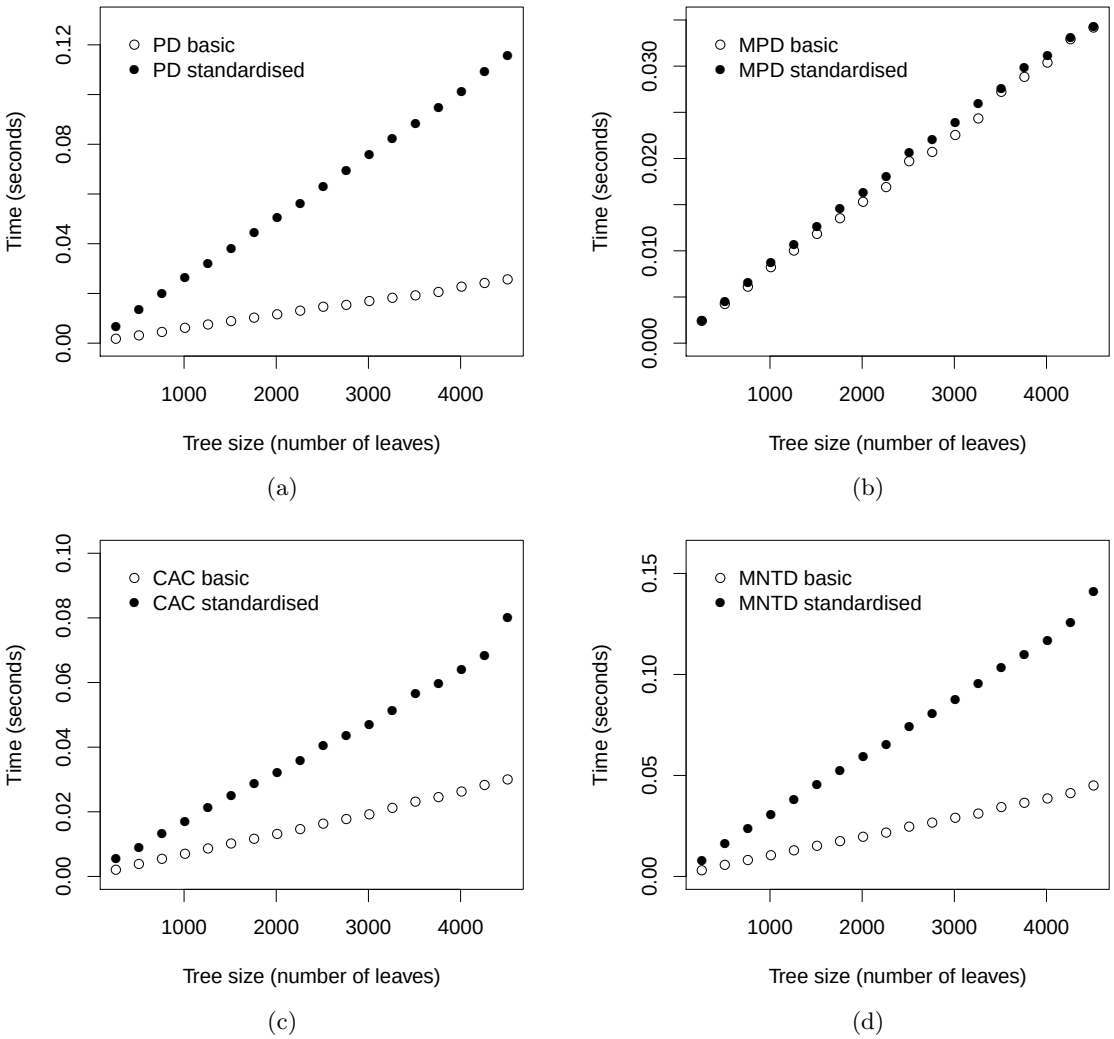



Figure 1: Running times of the query functions for all single-sample measures available in `PhyloMeasures`. For each query function and for each tree size, the figures illustrate the time that it takes for the function to process a set of twenty query samples.

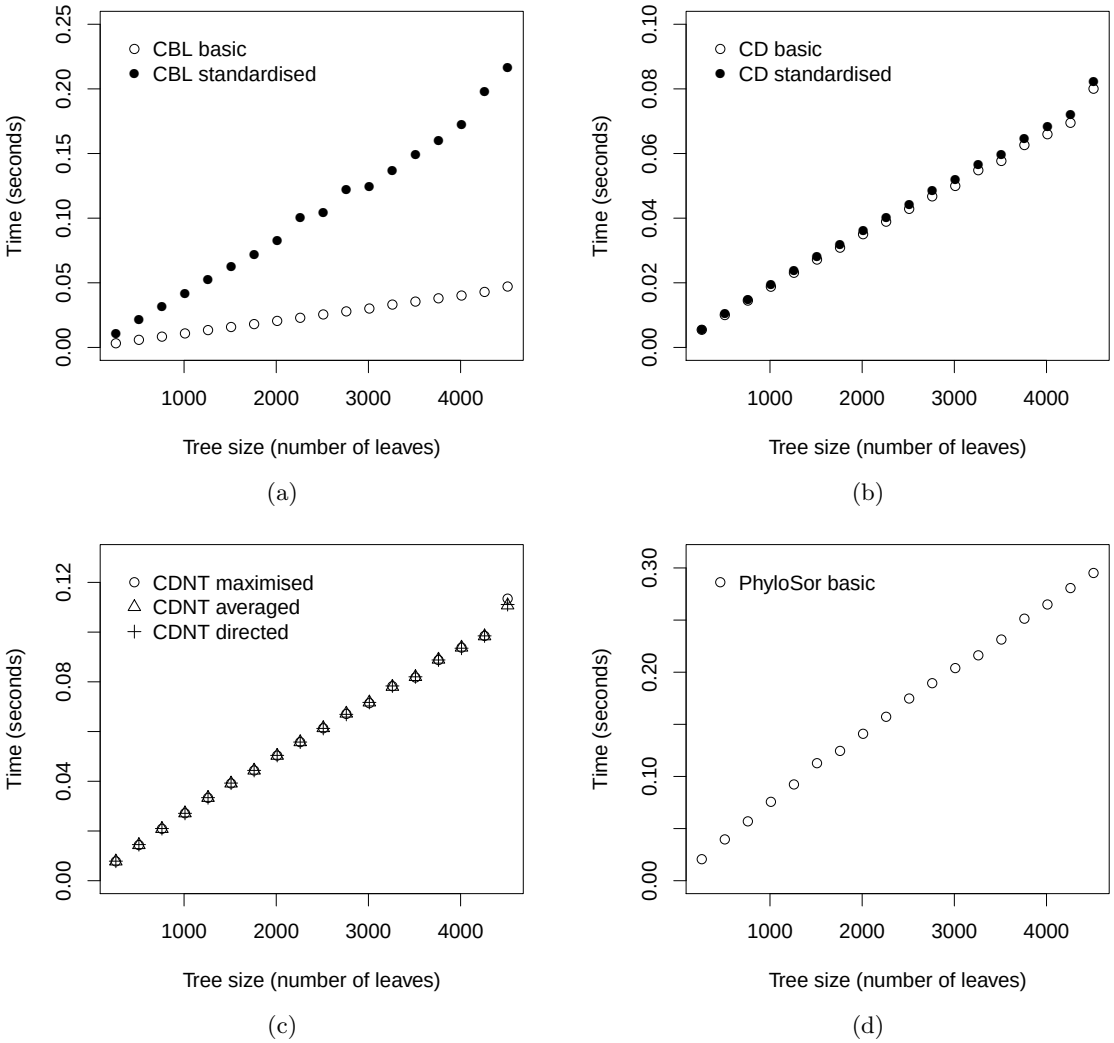
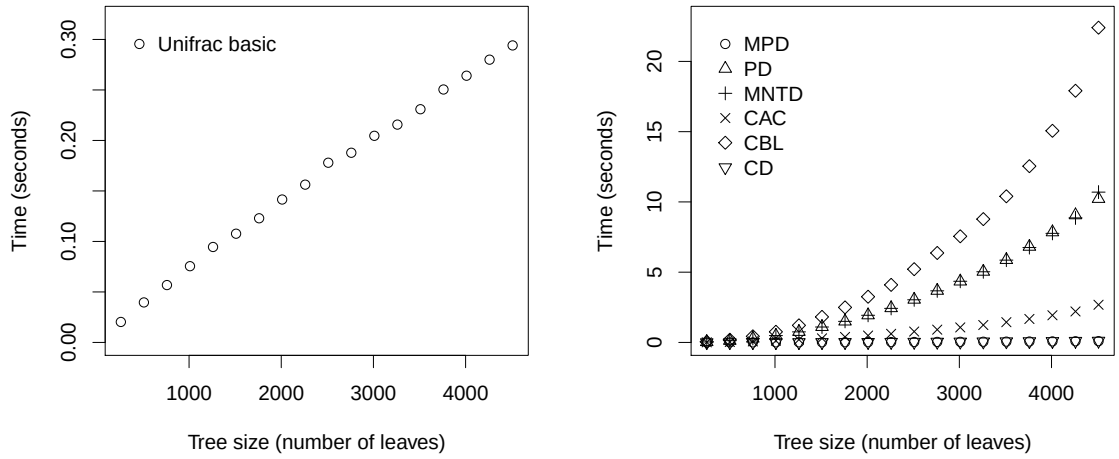



Figure 2: The running times of the query functions in `PhyloMeasures` for measures CBL, CD, CDNT and PhyloSor. For each query function and for each tree size, the figures illustrate the time that it takes for the function to process a set of twenty sample pairs.




(a) (b)

Figure 3: Running times for query and moment functions in `PhyloMeasures` measured on trees of different sizes. (a): The running times of the query function that computes the basic value of the UniFrac measure. For each tree size, the figure illustrates the time that it takes for the function to process a set of twenty sample pairs. (b): The running times of all available moment functions in `PhyloMeasures`. For a tree $\mathcal{T}$ that has $n$ leaves, the figure illustrates the time that it takes for a moment function to compute both the expectation and the deviation of a measure on $\mathcal{T}$ *for all* sample sizes in the range $\{1, 2, \ldots, \lfloor n/2 \rfloor\}$.

We see that all the examined functions perform very fast even for the largest trees in $\mathcal{U}$. It is evident that all functions in `PhyloMeasures` have a performance that scales very well as the size of the input tree increases. For the complete `mammals` tree (which consists of 4,510 leaves), the function which computes the basic MPD value takes 0.03 seconds, the function which computes the basic PD value 0.02 seconds, the standardised MPD function 0.034 seconds, and the standardised PD function 0.11 seconds. Note that each of these running times represents the total time taken for processing twenty samples of species.

We also tested the performance of our package on even larger trees. For this reason, we generated an artificial tree $\mathcal{T}$ of $10^5$ leaves using a randomised pure birth process. For this tree, we generated a set of twenty randomly selected leaf samples, and we measured the total time taken for computing the basic and standardised values of all supported measures on these samples. Each of the samples that we generated has size $10^5/k$, where $k$ ranges from one to twenty. The results of these experiments are shown in Table 2. Observe that each of the functions can process twenty queries on the large tree in less than 17 seconds. These results show that the functions in `PhyloMeasures` can handle efficiently even very large phylogenetic trees. This concludes our argument that `PhyloMeasures` is a very efficient tool for phylogenetic case studies.

Table 2: The time taken by each query function in `PhyloMeasures` for processing a set of twenty leaf samples on a phylogenetic tree of $10^5$ leaves. Times appear in seconds. We use "-" to indicate that a function is not available for some measure.

| Measure | Basic Value | Standardised Value |
|---|---|---|
| Phylogenetic Diversity | 0.97 | 7.74 |
| Mean Pairwise Distance | 1.45 | 1.54 |
| Mean Nearest Taxon Distance | 1.83 | 8.06 |
| Core Ancestor Cost | 1.19 | 5.36 |
| Common Branch Length | 1.93 | 15.99 |
| Community Distance | 3.67 | 3.80 |
| Community Distance Nearest Taxon | 4.29 | - |
| Phylogenetic Sorensen's Similarity | 4.29 | - |
| Unique Fraction | 4.41 | - |

We continue with presenting the interface for both versions of `PhyloMeasures` package. Next section provides a description of the R package, and Section 4 contains a description of the C++ package.

# 3 The R package

## 3.1 The Structure of the Package

The R package is a set of functions. The package contains one query function for each supported measure, except the CDNT for which three query functions are available; there is one function available for each version of this measure. These are the maximised, averaged, and directed CDNT. Also, the package provides one moment function for every measure except CDNT, PhyloSor, and UniFrac.

Each query function (except the ones related to CDNT, PhyloSor, and UniFrac) can be used to compute either the basic or the standardised values of the corresponding measure. More specifically, each such function receives a logical (boolean) argument (`is.standardised`); if this argument is set to `TRUE` then, for a given set of input samples, the standardised values of this measure are returned. Otherwise, the basic value of this measure is returned for each sample. Table 3 shows the names of the query and moment functions that are available in the R package for each supported measure.

Table 3: The names of the functions in the R package that are available for each supported measure.

| Measure | Query functions | Moment functions |
|---|---|---|
| Phylogenetic Diversity | `pd.query` | `pd.moments` |
| Mean Pairwise Distance | `mpd.query` | `mpd.moments` |
| Mean Nearest Taxon Distance | `mntd.query` | `mntd.moments` |
| Core Ancestor Cost | `cac.query` | `cac.moments` |
| Common Branch Length | `cbl.query` | `cbl.moments` |
| Community Distance | `cd.query` | `cd.moments` |
| Community Distance Nearest Taxon | `cdnt.query`<br>`cdnt.averaged.query`<br>`cdnt.directed.query` | - |
| Phylogenetic Sorensen's Similarity | `phylosor.query` | - |
| Unique Fraction | `unifrac.query` | - |

Next, we describe in detail the interface of the query functions in the R package. Then, in Section 3.3, we present the interface of the moment functions in the package.

## 3.2 Query Functions

```
pd.query(tree, matrix, is.standardised = FALSE)
```

**Description:** Calculates the basic or standardised value of the unrooted Phylogenetic Diversity (PD) measure for several samples of tips on a given phylogenetic tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `matrix` is a matrix with 0/1 values, where each row represents a tip sample. Each column name in the matrix must match a tip label on the input tree. If not all values in the matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the PD. For each input tip sample $R$, we derive

the standardised value of the PD from the basic value by subtracting the mean PD and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all tip samples that have the same number of elements as $R$, the tip sample whose value we want to standardise. This argument is optional, and its default value is set to `FALSE`.

The function returns a vector which stores the computed (standardised or basic) PD values. The $i$-th entry in this vector stores the (standardised or basic) PD value of the i-th tip sample in the input matrix.

---

`mpd.query(tree, matrix, is.standardised = FALSE)`

---

**Description:** Calculates the basic or standardised value of the Mean Pairwise Distance (MPD) measure for several samples of tips on a given phylogenetic tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `matrix` is a matrix with 0/1 values, where each row represents a tip sample. Each column name in the matrix must match a tip label on the input tree. If not all values in the matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the MPD. For each input tip sample $R$, we derive the standardised value of the MPD from the basic value by subtracting the mean MPD and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all tip samples that have the same number of elements as $R$, the tip sample whose value we want to standardise. This argument is optional, and its default value is set to `FALSE`.

The function returns a vector which stores the computed (standardised or basic) MPD values. The $i$-th entry in this vector stores the (standardised or basic) MPD value of the i-th tip sample in the input matrix.

---

`mntd.query(tree, matrix, is.standardised = FALSE)`

---

**Description:** Calculates the basic or standardised value of the Mean Nearest Taxon Distance (MNTD) measure for several samples of tips on a given phylogenetic tree. **Note:** in case the standardised values of the MNTD are computed, this function returns a result only if the input tree is ultrametric.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `matrix` is a matrix with 0/1 values, where each row represents a tip sample. Each column name in the matrix must match a tip label on the input tree. If not all values in the matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the MNTD. For each input tip sample $R$, we derive the standardised value of the MNTD from the basic value by subtracting the mean MNTD and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all tip samples that have the same number of elements as $R$, the tip sample whose value we want to standardise. If `is.standardised` is set to `TRUE`, the function returns a result only if the input tree is ultrametric. This argument is optional, and its default value is set to `FALSE`.

The function returns a vector which stores the computed (standardised or basic) MNTD values. The $i$-th entry in this vector stores the (standardised or basic) MNTD value of the i-th tip sample in the input matrix.

```
cac.query(tree, matrix, chi, is.standardised = FALSE)
```

**Description:** Calculates the basic or standardised value of the Core Ancestor Cost (CAC) measure for several samples of tips on a given phylogenetic tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `matrix` is a matrix with 0/1 values, where each row represents a tip sample. Each column name in the matrix must match a tip label on the input tree. If not all values in the matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. The input argument `chi` provides the value for the parameter $\chi$ that is used by the CAC measure–see Section 6 for the formal definition of the CAC. Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the CAC. For each input tip sample $R$, we derive the standardised value of the CAC from the basic value by subtracting the mean CAC and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all tip samples that have the same number of elements as $R$, the tip sample whose value we want to standardise. This argument is optional, and its default value is set to `FALSE`.

The function returns a vector which stores the computed (standardised or basic) CAC values. The $i$-th entry in this vector stores the (standardised or basic) CAC value of the i-th tip sample in the input matrix.

```
cbl.query(tree, matrix.a, matrix.b = NULL,
          query.matrix = NULL, is.standardised = FALSE)
```

**Description:** Calculates the basic or standardised value of the Common Branch Length (CBL) for several pairs of tip sets on a given phylogenetic tree. The Common Branch Length is the beta diversity version of Phylogenetic Diversity (PD), giving the sum of the lengths for the branches shared between two communities.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the (basic or standardised) CBL values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the CBL value. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes the (basic or standardised) CBL value between the $k$-th row of `matrix.a` and the $r$-th row of `matrix.b`. If `matrix.b` is not given, the function computes the CBL value between the $k$-th and $r$-th row of `matrix.a`. Argument `query.matrix` is optional.

Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the CBL. For each pair of tip samples $A$ and $B$, we derive the standardised value of the CBL from the basic value by subtracting the mean CBL and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all

tip samples such that one sample has $|A|$ tips, and the other sample has $|B|$ tips. This argument is optional, and its default value is set to `FALSE`.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the (basic or standardised) CBL values for all pairs of rows (tip sets) in `matrix.a` . If `matrix.b` is given but not `query.matrix`, the function computes the CBL values for all combinations of a row in `matrix.a` with rows in `matrix.b`. If `query.matrix` is given and `matrix.b` is not, the function returns the CBL values for the pairs of rows in `matrix.a` specified by `query.matrix`. If `query.matrix` and `matrix.b` are both given, CBL values are computed for the rows in `matrix.a` specified by the first column of `query.matrix` against the rows in `matrix.b` specified in the second column of `query.matrix` .

The function returns the (basic or standardised) CBL values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the CBL value for the pair of tip sets indicated in the $i$-th row of `query.matrix` . If `query.matrix` is not provided, the CBL values are returned in a matrix object; entry `[i,j]` in the output matrix stores the CBL value between the tip sets specified on the $i$-th and $j$-th row of `matrix.a` (if `matrix.b` is not specified), or the CBL value between the $i$-th row of `matrix.a` and the $j$-th row of `matrix.b` (if `matrix.b` is specified).

---

```
cd.query(tree, matrix.a, matrix.b = NULL,
         query.matrix = NULL, is.standardised = FALSE)
```

---

**Description:** Calculates the basic or standardised value of the Community Distance (CD) for several pairs of tip sets on a given phylogenetic tree. The Community Distance is the beta diversity version of the Mean Pairwise Distance (MPD), giving the average phylogenetic distance between two communities of species.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the (basic or standardised) CD values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the CD value. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes the (basic or standardised) CD value between the $k$-th row of `matrix.a` and the $r$-th row of `matrix.b`. If `matrix.b` is not given, the function computes the CD value between the $k$-th and $r$-th row of `matrix.a`. Argument `query.matrix` is optional.

Argument `is.standardised` is a logical (boolean) value that specifies whether the function returns the standardized or the basic value of the CD. For each pair of tip samples $A$ and $B$, we derive the standardised value of the CD from the basic value by subtracting the mean CD and dividing by the standard deviation of this measure. The mean and standard deviation are calculated among all tip samples such that one sample has $|A|$ tips, and the other sample has $|B|$ tips. This argument is optional, and its default value is set to `FALSE`.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the (basic or standardised) CD values for all pairs of rows (tip sets) in `matrix.a` . If

matrix.b is given but not query.matrix, the function computes the CD values for all combinations of a row in matrix.a with rows in matrix.b. If query.matrix is given and matrix.b is not, the function returns the CD values for the pairs of rows in matrix.a specified by query.matrix. If query.matrix and matrix.b are both given, CD values are computed for the rows in matrix.a specified by the first column of query.matrix against the rows in matrix.b specified in the second column of query.matrix .

The function returns the (basic or standardised) CD values for the requested pairs of tip sets. If query.matrix is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the CD value for the pair of tip sets indicated in the $i$-th row of query.matrix . If query.matrix is not provided, the CD values are returned in a matrix object; entry [i,j] in the output matrix stores the CD value between the tip sets specified on the $i$-th and $j$-th row of matrix.a (if matrix.b is not specified), or the CD value between the $i$-th row of matrix.a and the $j$-th row of matrix.b (if matrix.b is specified).

---

```
cdnt.query(tree, matrix.a, matrix.b = NULL,
           query.matrix = NULL)
```

---

**Description:** Calculates the basic value of the maximised Community Distance Nearest Taxon (CDNT) for several pairs of tip sets on a given phylogenetic tree. The CDNT is the beta diversity version of the Mean Nearest Taxon Distance (MNTD), giving the average phylogenetic distance between a species in a community $A$ and its nearest neighbour in a community $B$. The maximised CDNT is derived by calculating the basic value of the directed CDNT from set $A$ to set $B$, and from set $B$ to set $A$, and taking the maximum of these two values. A version of this function that computes the standardised value of the measure is not yet available.

Argument tree is a phylogenetic tree object, built using the R package ape [12]. The objects matrix.a and matrix.b are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument matrix.b is optional.

Object query.matrix is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the basic maximised CDNT values. Each row in query.matrix indicates a pair of tip sets for which we want to compute the maximised CDNT value. Let $k$ and $r$ be the values that are stored in the $i$-th row of query.matrix, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If matrix.b is given, the function computes the basic maximised CDNT value between the $k$-th row of matrix.a and the $r$-th row of matrix.b. If matrix.b is not given, the function computes the maximised CDNT value between the $k$-th and $r$-th row of matrix.a. Argument query.matrix is optional.

Queries can be given in four ways. If neither matrix.b nor query.matrix are given, the function computes the basic maximised CDNT values for all pairs of rows (tip sets) in matrix.a . If matrix.b is given but not query.matrix, the function computes the maximised CDNT values for all combinations of a row in matrix.a with rows in matrix.b. If query.matrix is given and matrix.b is not, the function returns the maximised CDNT values for the pairs of rows in matrix.a specified by query.matrix. If query.matrix and matrix.b are both given, maximised CDNT values are computed for the rows in matrix.a specified by the first column of query.matrix against the rows in matrix.b specified in the second column of query.matrix .

The function returns the basic maximised CDNT values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the basic maximised CDNT value for the pair of tip sets indicated in the $i$-th row of `query.matrix` . If `query.matrix` is not provided, the maximised CDNT values are returned in a matrix object; entry `[i,j]` in the output matrix stores the maximised CDNT value between the tip sets specified on the $i$-th and $j$-th row of `matrix.a` (if `matrix.b` is not specified), or the maximised CDNT value between the $i$-th row of `matrix.a` and the $j$-th row of `matrix.b` (if `matrix.b` is specified).

---

```
cdnt.averaged.query(tree, matrix.a, matrix.b = NULL,
                    query.matrix = NULL)
```

---

**Description:** Calculates the basic value of the averaged Community Distance Nearest Taxon (CDNT) for several pairs of tip sets on a given phylogenetic tree. The CDNT is the beta diversity version of the Mean Nearest Taxon Distance (MNTD), giving the average phylogenetic distance between a species in a community $A$ and its nearest neighbour in a community $B$. The averaged CDNT is computed based on the values of the directed CDNT; let $A$ and $B$ be two tip sets in the input tree. Let dCDNT$(A, B)$ be the directed CDNT from $A$ to $B$, and let dCDNT$(B, A)$ be the directed CDNT from $B$ to $A$. Let also $|A|$ denote the number of elements in $A$, and $|B|$ the number of elements in $B$. The basic value of the averaged CDNT between these two tip sets is equal to:

$$\frac{|A| \cdot \text{dCDNT}(A, B) + |B| \cdot \text{dCDNT}(B, A)}{|A| + |B|} .$$

A version of this function that computes the standardised value of the measure is not yet available.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the basic averaged CDNT values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the averaged CDNT value. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes the basic averaged CDNT value between the $k$-th row of `matrix.a` and the $r$-th row of `matrix.b`. If `matrix.b` is not given, the function computes the averaged CDNT value between the $k$-th and $r$-th row of `matrix.a`. Argument `query.matrix` is optional.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the basic averaged CDNT values for all pairs of rows (tip sets) in `matrix.a` . If `matrix.b` is given but not `query.matrix`, the function computes the averaged CDNT values for all combinations of a row in `matrix.a` with rows in `matrix.b`. If `query.matrix` is given and `matrix.b` is not, the function returns the averaged CDNT values for the pairs of rows in `matrix.a` specified by `query.matrix`. If `query.matrix` and `matrix.b` are both given, averaged CDNT values are computed for the rows in `matrix.a` specified by the first column of `query.matrix` against the rows in `matrix.b` specified in the second column of `query.matrix` .

The function returns the basic averaged CDNT values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the basic averaged CDNT value for the pair of tip sets indicated in the $i$-th row of `query.matrix` . If `query.matrix` is not provided, the averaged CDNT values are returned in a matrix object; entry `[i,j]` in the output matrix stores the averaged CDNT value between the tip sets specified on the $i$-th and $j$-th row of `matrix.a` (if `matrix.b` is not specified), or the averaged CDNT value between the $i$-th row of `matrix.a` and the $j$-th row of `matrix.b` (if `matrix.b` is specified).

---

```
cdnt.directed.query(tree, matrix.a, matrix.b = NULL,
                    query.matrix = NULL)
```

---

**Description:** Calculates the basic value of the directed Community Distance Nearest Taxon (CDNT) for several pairs of tip sets on a given phylogenetic tree. The CDNT is the beta diversity version of the Mean Nearest Taxon Distance (MNTD), giving the average phylogenetic distance between a species in a community $A$ and its nearest neighbour in a community $B$. Note that, unlike the other two-sample measures included in the package, this measure is asymmetric; for a taxon $t$ in set $A$ the nearest neighbor in $B$ may be taxon $s$, but for taxon $s$ the nearest neighbor in $A$ might be a third taxon $u$. Therefore, for every input pair of tip sets $A$ and $B$, the function returns two values; the directed CDNT value from $A$ to $B$, and the directed CDNT value from $B$ to $A$. The version of this function that computes the standardised value of the measure is not yet available.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the basic directed CDNT values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the two directed CDNT values. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes both the basic directed CDNT value from the tip set in the $k$-th row of `matrix.a` to the tip set in the $r$-th row of `matrix.b`, and the basic directed CDNT value from the tip set in the $r$-th row of `matrix.b` to the tip set in the $k$-th row of `matrix.a` . If `matrix.b` is not given, the two values computed are the basic directed CDNT from the $k$-th to the $r$-th set of `matrix.a`, and the basic directed CDNT from the $r$-th to the $k$-th set of `matrix.a`. Argument `query.matrix` is optional.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the two basic directed CDNT values for all pairs of rows (tip sets) in `matrix.a` . If `matrix.b` is given but not `query.matrix`, the function computes the two basic directed CDNT values for all combinations of a row in `matrix.a` with rows in `matrix.b`. If `query.matrix` is given and `matrix.b` is not, the function returns the two directed CDNT values for the pairs of rows in `matrix.a` specified by `query.matrix`. If `query.matrix` and `matrix.b` are both given, the two directed CDNT values are computed for the rows in `matrix.a` specified by the first column of `query.matrix` against the rows in `matrix.b` specified in the second column of `query.matrix` .

The function returns the directed CDNT values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in a list that contains two elements; each element is a one-dimensional vector storing directed CDNT values. Let $k$ and $r$ be the values stored in the $i$-th row

of `query.matrix`. For the first vector in the returned list, the $i$-th element of the vector is the basic directed CDNT value from the tip set of the $k$-th row in `matrix.a` to the tip set of the $r$-th row in `matrix.a` (if `matrix.b` is not specified), or to the tip set of the $r$-th row of `matrix.b` (if this matrix is specified). For the second vector of the output list, the $i$-th element of this vector is the basic directed CDNT value from the tip set of the $r$-th row in `matrix.a` (if `matrix.b` is not specified), or from the tip set of the $r$-th row of `matrix.b` (if this matrix is specified) to the tip set of the $k$-th row in `matrix.a`.

If `query.matrix` is not provided and `matrix.b` is provided, the function returns a list which consists of two matrix objects; for the first matrix in this list, entry `[i,j]` stores the basic directed CDNT value from the tip set specified on the $i$-th row of `matrix.a` to the tip set in the $j$-th row of `matrix.b`. For the second matrix in the output list, entry `[i,j]` stores the basic directed CDNT value from the tip set specified on the $j$-th row of `matrix.b` to the tip set in the $i$-th row of `matrix.a`.

If neither `query.matrix` nor `matrix.b` are provided, the function returns a single matrix object such that matrix entry `[i,j]` stores the basic directed CDNT value from the tip set specified on the $i$-th row of `matrix.a` to the tip set in the $j$-th row of this matrix.

---

```
phylosor.query(tree, matrix.a, matrix.b = NULL,
               query.matrix = NULL)
```

---

**Description:** Calculates the basic value of the Phylogenetic Sorensen's Similarity (PhyloSor) for several pairs of tip sets on a given phylogenetic tree. A version of this function that computes the standardised value of the measure is not yet available.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the basic PhyloSor values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the PhyloSor value. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes the basic PhyloSor value between the $k$-th row of `matrix.a` and the $r$-th row of `matrix.b`. If `matrix.b` is not given, the function computes the PhyloSor value between the $k$-th and $r$-th row of `matrix.a`. Argument `query.matrix` is optional.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the basic PhyloSor values for all pairs of rows (tip sets) in `matrix.a` . If `matrix.b` is given but not `query.matrix`, the function computes the PhyloSor values for all combinations of a row in `matrix.a` with rows in `matrix.b`. If `query.matrix` is given and `matrix.b` is not, the function returns the PhyloSor values for the pairs of rows in `matrix.a` specified by `query.matrix`. If `query.matrix` and `matrix.b` are both given, PhyloSor values are computed for the rows in `matrix.a` specified by the first column of `query.matrix` against the rows in `matrix.b` specified in the second column of `query.matrix` .

The function returns the basic PhyloSor values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the basic PhyloSor value for the pair of tip sets indicated in the $i$-th row of `query.matrix` . If `query.matrix` is not provided, the PhyloSor values are returned in a matrix object; entry `[i,j]` in

the output matrix stores the PhyloSor value between the tip sets specified on the $i$-th and $j$-th row of `matrix.a` (if `matrix.b` is not specified), or the PhyloSor value between the $i$-th row of `matrix.a` and the $j$-th row of `matrix.b` (if `matrix.b` is specified).

---

```
unifrac.query(tree, matrix.a, matrix.b = NULL,
              query.matrix = NULL)
```

---

**Description:** Calculates the basic value of the Unique Fraction measure (UniFrac) for several pairs of tip sets on a given phylogenetic tree. A version of this function that computes the standardised value of the measure is not yet available.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The objects `matrix.a` and `matrix.b` are matrices with 0/1 values, where each matrix row represents a tip sample. In each matrix, the name of every column must match a tip label on the input tree. If not all values in a matrix are binary, we consider two cases; if the matrix contains only non-negative values, all values are coerced to binary ones and a warning message is printed. If the matrix contains at least one negative value, the function throws an error. Argument `matrix.b` is optional.

Object `query.matrix` is a two-column matrix specifying the pairs of rows (tip sets) for which the function computes the basic UniFrac values. Each row in `query.matrix` indicates a pair of tip sets for which we want to compute the UniFrac value. Let $k$ and $r$ be the values that are stored in the $i$-th row of `query.matrix`, where $k$ is the value stored in the first column and $r$ is the value stored in the second column. If `matrix.b` is given, the function computes the basic UniFrac value between the $k$-th row of `matrix.a` and the $r$-th row of `matrix.b`. If `matrix.b` is not given, the function computes the UniFrac value between the $k$-th and $r$-th row of `matrix.a`. Argument `query.matrix` is optional.

Queries can be given in four ways. If neither `matrix.b` nor `query.matrix` are given, the function computes the basic UniFrac values for all pairs of rows (tip sets) in `matrix.a` . If `matrix.b` is given but not `query.matrix`, the function computes the UniFrac values for all combinations of a row in `matrix.a` with rows in `matrix.b`. If `query.matrix` is given and `matrix.b` is not, the function returns the UniFrac values for the pairs of rows in `matrix.a` specified by `query.matrix`. If `query.matrix` and `matrix.b` are both given, UniFrac values are computed for the rows in `matrix.a` specified by the first column of `query.matrix` against the rows in `matrix.b` specified in the second column of `query.matrix` .

The function returns the basic UniFrac values for the requested pairs of tip sets. If `query.matrix` is provided, then the values are returned in an one-dimensional vector. The $i$-th element of this vector is the basic UniFrac value for the pair of tip sets indicated in the $i$-th row of `query.matrix` . If `query.matrix` is not provided, the UniFrac values are returned in a matrix object; entry `[i,j]` in the output matrix stores the UniFrac value between the tip sets specified on the $i$-th and $j$-th row of `matrix.a` (if `matrix.b` is not specified), or the UniFrac value between the $i$-th row of `matrix.a` and the $j$-th row of `matrix.b` (if `matrix.b` is specified).

### 3.3 Moment Functions

```
pd.moments(tree, sample.sizes, comp.expectation = TRUE,
           comp.deviation = TRUE)
```

**Description:** Calculates the mean and standard deviation of the unrooted Phylogenetic Diversity (PD) for several tip set sizes on a given tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `sample.sizes` is a vector that contains non-negative integers; these are the tip set sizes for which we want to calculate the moments. The logical (boolean) argument `comp.expectation` specifies whether the function should compute the mean for each of the input set sizes. Similarly, argument `comp.deviation` specifies whether the function should compute the standard deviation for each input set size. Both arguments `comp.expectation` and `comp.deviation` are optional, and their default value is set to `TRUE`.

If both `comp.expectation` and `comp.deviation` are `TRUE`, the function returns a two-column matrix with one row per element in `sample.sizes`. The first column in this matrix stores the mean PD, and the second column stores the standard deviation of the measure for the corresponding sample size. If only one of `comp.expectation` or `comp.deviation` is `TRUE`, the function returns a vector with the corresponding values instead.

```
mpd.moments(tree, sample.sizes, comp.expectation = TRUE,
            comp.deviation = TRUE)
```

**Description:** Calculates the mean and standard deviation of the Mean Pairwise Distance (MPD) for several tip set sizes on a given tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `sample.sizes` is a vector that contains non-negative integers; these are the tip set sizes for which we want to calculate the moments. The logical (boolean) argument `comp.expectation` specifies whether the function should compute the mean for each of the input set sizes. Similarly, argument `comp.deviation` specifies whether the function should compute the standard deviation for each input set size. Both arguments `comp.expectation` and `comp.deviation` are optional, and their default value is set to `TRUE`.

If both `comp.expectation` and `comp.deviation` are `TRUE`, the function returns a two-column matrix with one row per element in `sample.sizes`. The first column in this matrix stores the mean MPD, and the second column stores the standard deviation of the measure for the corresponding sample size. If only one of `comp.expectation` or `comp.deviation` is `TRUE`, the function returns a vector with the corresponding values instead.

```
mntd.moments(tree, sample.sizes, comp.expectation = TRUE,
             comp.deviation = TRUE)
```

**Description:** Calculates the mean and standard deviation of the Mean Nearest Taxon Distance (MNTD) for several tip set sizes on a given tree. **Note:** this function returns a result only if the input tree is ultrametric.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The object `sample.sizes` is a vector that contains non-negative integers; these are the tip set sizes for which

we want to calculate the moments. The logical (boolean) argument `comp.expectation` specifies whether the function should compute the mean for each of the input set sizes. Similarly, argument `comp.deviation` specifies whether the function should compute the standard deviation for each input set size. Both arguments `comp.expectation` and `comp.deviation` are optional, and their default value is set to `TRUE`.

If both `comp.expectation` and `comp.deviation` are `TRUE`, the function returns a two-column matrix with one row per element in `sample.sizes`. The first column in this matrix stores the mean MNTD, and the second column stores the standard deviation of the measure for the corresponding sample size. If only one of `comp.expectation` or `comp.deviation` is `TRUE`, the function returns a vector with the corresponding values instead.

```
cac.moments(tree, chi, sample.sizes, k=2)
```

**Description:** Calculates the `k` first statistical moments of the Core Ancestor Cost (CAC) given $\chi$, for several tip set sizes on an input tree.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The argument `chi` corresponds to parameter $\chi$ of the CAC measure. It must be a real number in the interval $(0.5, 1]$. The object `sample.sizes` is a vector that contains non-negative integers; these are the tip set sizes for which we want to calculate the moments of the CAC. Argument `k` is a positive integer specifying the number of moments to compute. This argument is optional, and its default value is two.

The function returns a $k$-column matrix with as many as `length(sample.sizes)` rows. Entry `[i,j]` in the matrix gives the $j$-th moment for the $i$-th sample size in `sample.sizes`. The first moment in each row is the mean, and for $j$ larger than one the $j$-th returned moment is the centralised statistical moment of order $j$. Given a phylogenetic tree $\mathcal{T}$, a sample size $r$ and parameter $\chi$, the centralised statistical moment of order $k$ for the CAC is defined as:

$$E_{R\in \mathrm{Sub}(\mathcal{T},r)}[(\mathrm{CAC}_\chi(\mathcal{T},R) - \mu_{\mathrm{CAC},\chi}(\mathcal{T},r))^k] \ ,$$

where $E_{R\in \mathrm{Sub}(\mathcal{T},r)}$ denotes the expected value of a random variable for all tip sets in $\mathcal{T}$ that consist of $r$ tips each, and $\mu_{\mathrm{CAC},\chi}(\mathcal{T},r)$ indicates the mean value of the CAC over all these sets.

```
cbl.moments(tree, sample.sizes, comp.expectation = TRUE,
            comp.deviation = TRUE)
```

**Description:** Calculates the mean and standard deviation of the Common Branch Length (CBL) on a given tree for several pairs of set sizes. The CBL is the beta diversity version of Phylogenetic Diversity (PD), giving the total branch length shared between two communities.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The argument `sample.sizes` is a two-column matrix of non-negative integers indicating the tip set sizes for which the moments should be calculated. The moments are calculated for each row of the matrix. Let $x$ and $y$ be the values that are stored in the $i$-th row of `sample.sizes`. For this row, the CBL moments are calculated among all pairs of tip subsets in the input tree such that one set has $x$ elements and the other set has $y$ elements. The logical (boolean) argument `comp.expectation` specifies whether the function should compute the mean for each of the input set sizes. Similarly, argument `comp.deviation` specifies whether the function should compute the standard deviation for each input set size. Both

arguments `comp.expectation` and `comp.deviation` are optional, and their default value is set to `TRUE`.

If both `comp.expectation` and `comp.deviation` are `TRUE`, the function returns a two-column matrix with one row per element in `sample.sizes`; the first column of the output matrix stores the mean CBL and the second column stores the standard deviation for this measure. If only one of `comp.expectation` or `comp.deviation` is `TRUE`, the function returns a vector with the corresponding values instead.

```
cd.moments(tree, sample.sizes, comp.expectation = TRUE,
           comp.deviation = TRUE)
```

**Description:** Calculates the mean and standard deviation of the Community Distance (CD) on a given tree for several pairs of set sizes. The CD is the beta diversity version of the Mean Pairwise Distance (MPD), giving the average phylogenetic distance between two species communities.

Argument `tree` is a phylogenetic tree object, built using the R package `ape` [12]. The argument `sample.sizes` is a two-column matrix of non-negative integers indicating the tip set sizes for which the moments should be calculated. The moments are calculated for each row of the matrix. Let $x$ and $y$ be the values that are stored in the $i$-th row of `sample.sizes`. For this row, the CD moments are calculated considering all pairs of tip sets that can be extracted from the input tree such that one set has $x$ elements and the other set has $y$ elements. The logical (boolean) argument `comp.expectation` specifies whether the function should compute the mean for each of the input set sizes. Similarly, argument `comp.deviation` specifies whether the function should compute the standard deviation for each input set size. Both arguments `comp.expectation` and `comp.deviation` are optional, and their default value is set to `TRUE`.

If both `comp.expectation` and `comp.deviation` are `TRUE`, the function returns a two-column matrix with one row per element in `sample.sizes`; the first column of the output matrix stores the mean CD and the second column stores the standard deviation for this measure. If only one of `comp.expectation` or `comp.deviation` is `TRUE`, the function returns a vector with the corresponding values instead.

# 4 The C++ package

The C++ package is developed based on the object-oriented programming paradigm. All important concepts (phylogenetic tree, measures) are represented by object classes, and the main functions of the package are available as members of these classes. All class types are included in a central class that we call the *kernel*.

## 4.1 The Kernel

The kernel is the class that contains all types needed for using the package functions. Among other types, the kernel contains one class type for each phylogenetic measure supported in the package. We call these class types the *measure classes*. Each measure class contains the query and moment functions for this measure. The name of the kernel class is `Phylogenetic_measures_kernel`. This class is a *template*; to define this class we need to provide another type as a parameter. The template parameter of the kernel is a class that describes what kind of arithmetic should be used by the package functions. We call this class the *numeric traits*. The numeric traits contain the number type that will be used for all numeric operations. In this way, the user can select the number type that matches his needs, and tune the functions in the package to perform operations which are as precise as he desires. The user can do that by designing his own numeric traits class and feeding this class to the kernel as the template parameter. In Section 4.7 we describe how the user can design a simple numeric traits class. If the user does not specify a traits class as a kernel parameter, a default traits class is used which handles the built-in C++ `double` number type. Next, we provide a simple program that shows how to use a kernel and execute a measure's moment function.

## 4.2 A Simple Program

Suppose that we want to compute the expectation of the PD for a given phylogenetic tree $\mathcal{T}$ and for sample size $s = 5$. Suppose also that the tree is stored in the text file `tree.tre`, written in the standard Newick tree format. To compute the expectation of the PD for this tree, we have to run the following simple program:

```
1  #include <iostream>
2  #include "Phylogenetic_measures_kernel.h"
3
4  typedef Phylogenetic_measures_kernel<>     Kernel;
5  typedef Kernel::Number_type                Number_type;
6  typedef Kernel::Phylogenetic_diversity     Phylogenetic_diversity;
7  typedef Phylogenetic_diversity::Tree_type  Tree_type;
8
9  int main(void)
10 {
11   Tree_type tree;
12
13   tree.construct_from_file("tree.tre");
14
15   Phylogenetic_diversity pd(tree);
16
17   int sample_size = 5;
18   Number_type mean = pd.compute_expectation(sample_size);
19
20   std::cout << std::endl << " The mean of the PD for sample size 5 is: " << mean
21             << std::endl << std::endl;
22
23   return 0;
24 }
```

That will work assuming that the tree file is stored in the same directory as the above C++ program. We now have a close look at this program, and explain what happens there line-by-line. In the two first lines we have the statements for including all libraries needed by the program. The only file that we have to include from the `PhyloMeasures` source code is `Phylogenetic_measures_kernel.h`, which contains the description of the kernel. Then, in line 4 we define the type of the kernel that we want to use. There we see that the kernel type does not have any additional template parameter. As mentioned earlier in this section, when we do not specify a parameter for the kernel then by default all classes and functions provided by this kernel will perform mathematical operations using the C++ built-in `double` type. This is the case for the `Kernel` type that we define.

Next, in lines 5-7 we export from the kernel all the types that we want to use in the program. First we export class `Number_type`, which is obviously the number type that will be used for all mathematical operations. As we did not specify any traits class as a kernel parameter, this type is set by default to the C++ `double` type. Then, we export class `Phylogenetic_diversity` which contains the query and moment functions of the PD measure. From this class we export the type `Tree_type` which will be used for representing the phylogenetic tree that we want to process. The kernel class contains one class for every phylogenetic measure, and every measure class contains its own tree type. All these tree types have the same interface. Each measure class provides its own tree type because, internally, each tree class stores different kinds of data, to speed up the execution of the measures functions.

After exporting the types that we need, we continue with building the tree. In lines 11-13 we construct an (initially empty) tree object, and then we input to this object the phylogenetic tree that is stored in file `tree.tre`. This file must be a text file, and the tree that is stored there should be described in the standard Newick tree format.

Then, in line 15 we construct an object of the `Phylogenetic_diversity` class. The query and moments functions of the PD measure are members of this object. To construct the object we pass as an input argument the tree that we built. From thereon, all functions that are called on this measure object will use that tree. In lines 17-18 we call the member function of the `Phylogenetic_diversity` object which computes the expectation of the PD. The output returned by the function is an object of type `Number_type`.

This simple program is available with the package in folder `PhyloMeasures/Examples`, in the file with name `compute_PD_expectation.cpp`. In this folder we provide also other example programs that show e.g. how to call the query function of a measure. Also, in folder `PhyloMeasures/Programs` we provide several useful programs which exploit the full potential of the C++ package. The user can execute these programs from the command line, and call any query or moment function with a single command. All these programs use `double` floating point arithmetic. The command line interface of each program is presented in Section 4.8.

Next, we describe in detail the interface of all classes that are available in the package. These include the classes of the phylogenetic measures, the classes that represent phylogenetic trees, and the kernel class.

## 4.3 The Interface of the Kernel Class and Included Types

The kernel class is a template class of the type:

```
template< class Numeric_traits = Numeric_traits_double>
struct Phylogenetic_measures_kernel { ... };
```

As described already, the optional template parameter of this class is the numeric traits type that defines the arithmetic used by the package functions. The default parameter is a traits class that uses double precision floats. The kernel exports from the numeric traits the `Number_type` class which is used in the numeric operations of the package functions. The kernel also contains one class for each phylogenetic measure that is available in the package. These are the following classes:

- `Phylogenetic_diversity`
- `Mean_pairwise_distance`
- `Mean_nearest_taxon_distance`
- `Core_ancestor_cost`
- `Common_branch_length`
- `Community_distance`
- `Community_distance_nearest_taxon`
- `Phylogenetic_Sorensens_similarity`
- `Unique_fraction`

The kernel itself does not contain any member functions.

## 4.4 The Measure Classes

Each measure class provides several member functions; these are the query and moment functions for the corresponding measure. Every measure class contains also a type that represents the phylogenetic tree objects that are handled by this class. For a measure class `X`, this type is `X::Tree_type`. Each measure class contains its own tree type, and the user should be careful so as not to use tree objects of measure class `X` with the functions of another measure class `Y`.

To construct an object of a measure class `X`, we need to provide an object of type `X::Tree_type` (see also the example program in Section 4.2). After doing this, we do not need to provide any more tree data. All query and moment functions which are members of this object will use the tree that was provided at the object's construction.

All measure classes provide member functions that execute queries for given samples of species. Every function of this kind can handle multiple queries with a single call. For example, any single-sample measure class has a member function with the following interface:

```
template <class OutputIterator>
csv_matrix_query_basic(char *filename, OutputIterator ot );
```

This function computes the basic value of the measure for a set of query leaf samples. The query samples are stored in file `filename`. This is a `csv` file which stores the query samples in a matrix format; each column in the matrix corresponds to a different species, and each row describes a different input sample (except the first row which stores the species names). If the $i$-th input sample includes the species represented by the $j$-th column, then the corresponding entry in the matrix is 1 (or any other positive value), otherwise it is zero. A complete description of this file format can be found in Section 4.5.1. The second argument of the function is an output iterator. The function computes the basic value of the measure for every query sample that appears in `filename`, and the results are returned through the output iterator. The following example shows how to call this function for the MPD measure:

```
1  #include <iostream>
2  #include "Phylogenetic_measures_kernel.h"
3
4  typedef Phylogenetic_measures_kernel<>     Kernel;
5  typedef Kernel::Number_type                Number_type;
6  typedef Kernel::Mean_pairwise_distance     Mean_pairwise_distance;
7  typedef Mean_pairwise_distance::Tree_type  Tree_type;
8
9  int main(void)
10 {
11   Tree_type tree;
12
13   tree.construct_from_file("tree.tre");
14
15   Mean_pairwise_distance mpd(tree);
16
17   std::vector<Number_type> results;
18   char matrix_filename[] = "example_matrix.csv";
19   mpd.csv_matrix_query_basic(matrix_filename, std::back_inserter(results));
20
21   std::cout << std::endl << " The MPD values are the following: " << std::endl;
22   std::cout << "---------------------------------" << std::endl;
23
24   for( size_t i=0; i<results.size(); i++)
25     std::cout << results[i] << std::endl;
26
27   std::cout << std::endl;
28
29   return 0;
30 }
```

In the above example, we create an object of the `Mean_pairwise_distance` class, and then we call the query member function of this object to compute the basic MPD values of the samples in file `example_matrix.csv`. The computed values are stored in an `std::vector` object since we pass the back insert iterator of this object to the query function. The above example is available with the package in the file `PhyloMeasures/Examples/compute_MPD_matrix_query.cpp`.

Every measure class also has a member function that computes the standardised values for several query samples (except the classes related to measures CDNT, PhyloSor, and UniFrac). Two-sample measure classes provide functions for computing the basic and the standardised values of the measures between pairs of samples. The `Community_distance_nearest_taxon` measure class provides also a few more query member functions due to the measure's special nature–see Section 4.4.7 for more details.

For most measure classes there is a member function available for computing the expectation, the variance, and the deviation of this measure for a given sample size. This is not the case for classes `Community_distance_nearest_taxon`, `Phylogenetic_Sorensens_similarity` and `Unique_fraction` that do not provide any of these functions. Also, class `Core_ancestor_cost` contains an extra member function which, for a given sample size $s$ and positive integer $k$, computes all first $k$ moments of the CAC for this sample size.

Next, we present in detail the interface of all member functions that are available with each measure class. For some of these functions, the input arguments are files and matrices describing the queries that the function should process. The format of these arguments is explained in Section 4.5.

### 4.4.1 Phylogenetic_diversity

```
Phylogenetic_diversity(typename Phylogenetic_diversity::Tree_type &tree)
```

**Description:** Constructs an object of the `Phylogenetic_diversity` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template <class OutputIterator>
void csv_matrix_query_basic(char *filename, OutputIterator ot)
```

**Description:** Computes the basic value of the unrooted PD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in file `filename`. This file should be in the the matrix file format which is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the basic value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void csv_matrix_query_standardised(char *filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the unrooted PD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in file `filename`. This file should be in the matrix file format which is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the standardised value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_basic(std::vector<std::string> &names,
                        std::vector< std::vector<bool> > &matrix,
                        OutputIterator ot)
```

**Description:** Computes the basic value of the unrooted PD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by vectors `names` and `matrix`. These vectors should have the structure that is described in Section 4.5.2. The last argument is an output iterator. The function computes the basic value of the PD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_standardised(std::vector<std::string> &names,
                               std::vector< std::vector<bool> > &matrix,
                               OutputIterator ot)
```

**Description:** Computes the standardised value of the unrooted PD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the standardised value of the PD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin, RangeIterator rend)
```

**Description:** Returns the basic value of the unrooted PD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are represented by a range of iterators, and each iterator in this range refers to an object of type `std::string`. Each string in the input range must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename)
```

**Description:** Returns the basic value of the unrooted PD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are stored in file `filename`. The names appear in this file separated by commas and with no additional characters–see the definition of the species names file format in Section 4.5.1 for an example. Each name in the input file must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size)
```

**Description:** Returns the expectation of the unrooted PD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the mean.

```
Number_type compute_variance(int sample_size)
```

**Description:** Returns the variance of the unrooted PD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the variance.

```
Number_type compute_deviation(int sample_size)
```

**Description:** Returns the standard deviation of the unrooted PD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the deviation.

### 4.4.2 Mean_pairwise_distance

```
Mean_pairwise_distance(typename Mean_pairwise_distance::Tree_type &tree)
```

**Description:** Constructs an object of the `Mean_pairwise_distance` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template <class OutputIterator>
void csv_matrix_query_basic(char *filename, OutputIterator ot)
```

**Description:** Computes the basic value of the MPD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in the file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the basic value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void csv_matrix_query_standardised(char *filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the MPD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in the file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the standardised value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_basic(std::vector<std::string> &names,
                        std::vector< std::vector<bool> > &matrix,
                        OutputIterator ot)
```

**Description:** Computes the basic value of the MPD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is described in Section 4.5.2. The last argument is an output iterator. The function computes the basic value of the MPD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_standardised(std::vector<std::string> &names,
                               std::vector< std::vector<bool> > &matrix,
                               OutputIterator ot)
```

**Description:** Computes the standardised value of the MPD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the standardised value of the MPD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin, RangeIterator rend)
```

**Description:** Returns the basic value of the MPD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are represented by a range of iterators, and each iterator in this range refers to an object of type `std::string`. Each string in the input range must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename)
```

**Description:** Returns the basic value of the MPD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are stored in file `filename`. The names appear in this file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input file must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size)
```

**Description:** Returns the expectation of the MPD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the mean.

```
Number_type compute_variance(int sample_size)
```

**Description:** Returns the variance of the MPD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the variance.

```
Number_type compute_deviation(int sample_size)
```

**Description:** Returns the standard deviation of the MPD among all samples of `sample_size` leaves in the stored tree. Each of these samples contributes with the same weight to the computation of the deviation.

### 4.4.3 Mean_nearest_taxon_distance

```
Mean_nearest_taxon_distance(typename Mean_nearest_taxon_distance::Tree_type &tree)
```

**Description:** Constructs an object of the `Mean_nearest_taxon_distance` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template <class OutputIterator>
void csv_matrix_query_basic(char *filename, OutputIterator ot)
```

**Description:** Computes the basic value of the MNTD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in the file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the basic value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void csv_matrix_query_standardised(char *filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the MNTD measure for several samples of species in the stored tree $\mathcal{T}$. **This function has a normal execution only if $\mathcal{T}$ is ultrametric. Otherwise, no results are returned.** The input samples are listed in the file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the standardised value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_basic(std::vector<std::string> &names,
                        std::vector< std::vector<bool> > &matrix,
                        OutputIterator ot)
```

**Description:** Computes the basic value of the MNTD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the basic value of the MNTD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_standardised(std::vector<std::string> &names,
                               std::vector< std::vector<bool> > &matrix,
                               OutputIterator ot)
```

**Description:** Computes the standardised value of the MNTD measure for several samples of species in the stored tree $\mathcal{T}$. **This function has a normal execution only if $\mathcal{T}$ is ultrametric. Otherwise, no results are returned.** The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the standardised value of the MNTD for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin, RangeIterator rend)
```

**Description:** Returns the basic value of the MNTD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are represented by a range of iterators, and each iterator in this range refers to an object of type `std::string`. Each string in the input range name must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename)
```

**Description:** Returns the basic value of the MNTD measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are stored in the file `filename`. The names appear in the file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input file must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size)
```

**Description:** Returns the expectation of the MNTD among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the mean. **This function has a normal execution only if $\mathcal{T}$ is ultrametric. Otherwise, the returned value is equal to minus one.**

```
Number_type compute_variance(int sample_size)
```

**Description:** Returns the variance of the MNTD among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the variance. **This function has a normal execution only if $\mathcal{T}$ is ultrametric. Otherwise, the returned value is equal to minus one.**

```
Number_type compute_deviation(int sample_size)
```

**Description:** Returns the standard deviation of the MNTD among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the deviation. **This function has a normal execution only if $\mathcal{T}$ is ultrametric. Otherwise, the returned value is equal to minus one.**

#### 4.4.4 Core_ancestor_cost

```
Core_ancestor_cost(typename Core_ancestor_cost::Tree_type &tree, Number_type chi)
```

**Description:** Constructs an object of the `Core_ancestor_cost` class. The input argument `chi` corresponds to parameter $\chi$ that appears in the definition of the measure–see the exact description of the CAC in Section 6 for more details. The value of `chi` must belong to the interval $(0.5, 1.0]$, otherwise the function throws an error. The input tree and value `chi` are stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template <class OutputIterator>
void csv_matrix_query_basic(char *filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CAC measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the basic value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void csv_matrix_query_standardised(char *filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CAC measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are listed in file `filename`. This file should be in the matrix file format that is described in Section 4.5.1. The second argument of the function is an output iterator. The function computes the standardised value of the measure for every query sample in `filename`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_basic(std::vector<std::string> &names,
                        std::vector< std::vector<bool> > &matrix,
                        OutputIterator ot)
```

**Description:** Computes the basic value of the CAC measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the basic value of the CAC for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class OutputIterator>
void matrix_query_standardised(std::vector<std::string> &names,
                               std::vector< std::vector<bool> > &matrix,
                               OutputIterator ot)
```

**Description:** Computes the standardised value of the CAC measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are represented by the vectors `names` and `matrix`. These vectors should have the structure that is presented in Section 4.5.2. The last argument is an output iterator. The function computes the standardised value of the CAC for every query sample in `matrix`, and the results are returned through the output iterator.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin, RangeIterator rend)
```

**Description:** Returns the basic value of the CAC measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are represented by a range of iterators, and each iterator in this range refers to an object of type `std::string`. Each string in the input range must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename)
```

**Description:** Returns the basic value of the CAC measure for a given sample of species in the stored tree $\mathcal{T}$. The species names are stored in the file `filename`. The names appear in the file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input file must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size)
```

**Description:** Returns the expectation of the CAC among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the mean.

```
Number_type compute_variance(int sample_size)
```

**Description:** Returns the variance of the CAC among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the variance.

```
Number_type compute_deviation(int sample_size)
```

**Description:** Returns the standard deviation of the CAC among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$. Each of these samples contributes with the same weight to the computation of the deviation.

```
template<class OutputIterator>
void compute_first_k_raw_moments(int k, int sample_size, OutputIterator ot)
```

**Description:** Returns the first `k` raw moments of the CAC measure among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$, given the stored value `chi`. The $i$-th raw moment of the CAC for sample size $s$ given $\chi$ is defined as:

$$E_{\substack{S\in\mathcal{T}\\|S|=s}}\left[\mathrm{CAC}^i(\mathcal{T},S,\chi)\right] \ .$$

Here, we use $E_{S\in\mathcal{T},|S|=s}[X]$ to denote the expected value of random variable $X$ among all leaf samples in $\mathcal{T}$ that have size $s$.

```
template <class OutputIterator>
void compute_first_k_centralised_moments(int k,int sample_size,OutputIterator ot)
```

**Description:** Returns the first `k` moments of the CAC measure among all samples of `sample_size` leaves in the stored tree $\mathcal{T}$, given the stored value `chi`. The first returned moment is the expectation of this measure, and for $i$ larger than one, the $i$-th centralised moment of the CAC for sample size $s$ given $\chi$ is defined as:

$$E_{\substack{S\in\mathcal{T}\\|S|=s}}\left[(\mathrm{CAC}(\mathcal{T},S,\chi)-\mu_{\mathrm{CAC}}(\mathcal{T},s,\chi))^i\right] \ .$$

where $\mu_{\mathrm{CAC}}(\mathcal{T},s,\chi)$ is the expectation (mean) of the CAC given $\chi$ among all species samples in $\mathcal{T}$ of size $s$. We use $E_{S\in\mathcal{T},|S|=s}[X]$ to denote the expected value of random variable $X$ among all leaf samples in $\mathcal{T}$ that have size $s$. Every sample $S$ is picked uniformly out of all samples of leaves in $\mathcal{T}$ that have size $s$.

### 4.4.5 Common_branch_length

```
Common_branch_length(typename Common_branch_length::Tree_type &tree)
```

**Description:** Constructs an object of the `Common_branch_length` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_basic(char *matrix_filename_a,
                       char *matrix_filename_b,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the CBL for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair that consists of a sample in file `matrix_filename_a` and a sample in file `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_filename_a` consist of $n$ samples, and let the matrix in `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic CBL values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, next $m$ values are the basic values for pairs $(A_2, B_1)$, $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_standardised(char *matrix_filename_a,
                              char *matrix_filename_b,
                              OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is

a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the standardised CBL value for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the standardised value of the measure for every pair of samples that are described in the matrix files, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_filename_a` consist of $n$ samples, and let `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the standardised values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the standardised values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

---

```
template<class OutputIterator>
int csv_matrix_query_basic(char *matrix_filename, OutputIterator ot)
```

---

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix_filename`. The second argument of the function is an output iterator. The function computes the basic value of the CBL for every pair of samples in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic CBL values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in file `matrix_filename`.

---

```
template<class OutputIterator>
int csv_matrix_query_standardised(char *matrix_filename, OutputIterator ot)
```

---

**Description:** Computes the standardised value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the measure *for every pair* of samples that appear in `matrix_filename`. The second argument of the function is an output iterator. The function computes the standardised value of the CBL for every pair of samples that are described in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the standardised CBL values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the standardised values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the CBL measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the CBL measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_standardised
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the CBL measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the standardised value of the CBL measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the standardised CBL value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the CBL measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic value of the CBL measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples in file `matrix_filename`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_standardised
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the value of the CBL measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the value of the CBL are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the standardised CBL value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_basic(const std::vector<std::string> &names_a,
                   const std::vector<std::vector<bool> > &matrix_a,
                   const std::vector<std::string> &names_b,
                   const std::vector<std::vector<bool> > &matrix_b,
                   OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore,

if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the CBL for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic CBL values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic CBL values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_standardised(const std::vector<std::string> &names_a,
                          const std::vector<std::vector<bool> > &matrix_a,
                          const std::vector<std::string> &names_b,
                          const std::vector<std::vector<bool> > &matrix_b,
                          OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the standardised value of the CBL for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the standardised CBL value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the standardised CBL values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the standardised CBL values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_basic(const std::vector<std::string> &names,
                       const std::vector< std::vector<bool> > &matrix,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the CBL for every pair of samples in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic CBL values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
int matrix_query_standardised(const std::vector<std::string> &names,
                              const std::vector< std::vector<bool> > &matrix,
                              OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the standardised value of the CBL for every pair of samples in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the standardised CBL values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the standardised values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the CBL measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic CBL value are described in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_standardised
(const std::vector<std::string> &names_a,
 const std::vector< std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector< std::vector<bool> > &matrix_b,
 const std::vector< std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the CBL measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the standardised value of the CBL measure are described in `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the standardised CBL value for each sample pair listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the CBL measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the CBL measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic CBL value are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic CBL value for each sample pair that is described in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_standardised
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the standardised value of the CBL for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the CBL *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the standardised value of the CBL measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the standardised CBL value for each sample pair that is described in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin_a, RangeIterator rend_a,
                       RangeIterator rbegin_b, RangeIterator rend_b)
```

**Description:** Returns the basic value of the CBL measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of each sample are represented by a range of iterators; the species names of sample $A$ are represented by the range `[rbegin_a, rend_a)`, and the species names of $B$ are represented by the range `[rbegin_b, rend_b)`. Each iterator in these ranges refers to an object of type `std::string`. Every input string must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename_a, char* filename_b)
```

**Description:** Returns the basic value of the CBL measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of sample $A$ are stored in file `filename_a`, and the species names of sample $B$ are stored in file `filename_b`. The names appear in each file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input files must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size_a, int sample_size_b)
```

**Description:** Returns the expectation of the CBL among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these sample pairs contributes with the same weight to the computation of the mean.

```
Number_type compute_variance(int sample_size_a, int sample_size_b)
```

**Description:** Returns the variance of the CBL among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these pairs contributes with the same weight to the computation of the variance.

```
Number_type compute_deviation(int sample_size_a, int sample_size_b)
```

**Description:** Returns the standard deviation of the CBL among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these pairs contributes with the same weight to the computation of the deviation.

### 4.4.6 Community_distance

```
Community_distance(typename Community_distance::Tree_type &tree)
```

**Description:** Constructs an object of the `Community_distance` class. The input tree is stored in the constructed object, and will be used thereon by every member function for computing queries or moments of this measure.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_basic(char *matrix_filename_a,
                       char *matrix_filename_b,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the CD for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair that consists of a sample in file `matrix_filename_a` and a sample in file `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let the matrix in `matrix_filename_a` consist of $n$ samples, and let the matrix in `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic CD values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic CD values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_standardised(char *matrix_filename_a,
                              char *matrix_filename_b,
                              OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is

a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the standardised CD value for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the standardised value of the measure for every pair of samples that is described in the matrix files, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_filename_a` consist of $n$ samples, and let `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the standardised values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the standardised values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_basic(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is an output iterator. The function computes the basic value of the CD for every pair of samples in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic CD values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in the file `matrix_filename`.

```
template<class OutputIterator>
int csv_matrix_query_standardised(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is an output iterator. The function computes the standardised value of the CD for every pair of samples in the input file, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the standardised CD values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the standardised values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the CD measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the CD measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_standardised
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the CD measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the standardised CD value are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the standardised CD value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the CD measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic value of the CD measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_standardised
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the standardised value of the CD measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the standardised CD value are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the standardised CD value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_basic(const std::vector<std::string> &names_a,
                   const std::vector<std::vector<bool> > &matrix_a,
                   const std::vector<std::string> &names_b,
                   const std::vector<std::vector<bool> > &matrix_b,
                   OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the CD for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic CD values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic CD values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_standardised(const std::vector<std::string> &names_a,
                          const std::vector<std::vector<bool> > &matrix_a,
                          const std::vector<std::string> &names_b,
                          const std::vector<std::vector<bool> > &matrix_b,
                          OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the standardised value of the CD for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the standardised CD value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$

indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the standardised CD values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the standardised CD values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_basic(const std::vector<std::string> &names,
                       const std::vector<std::vector<bool> > &matrix,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the CD for every pair of samples that are described in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair that consists of the $i$-th and $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic CD values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic CD values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
int matrix_query_standardised(const std::vector<std::string> &names,
                              const std::vector<std::vector<bool> > &matrix,
                              OutputIterator ot)
```

**Description:** Computes the standardised value of the CD for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the standardised value of the CD for every pair of samples that are described in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair that consists of the $i$-th and $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the standardised CD values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the standardised values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the CD measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic value of the CD measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_standardised
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the standardised value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the CD measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the standardised value of the CD measure are listed in `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the standardised CD value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the CD measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the CD measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic value of the CD measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic CD value for each sample pair listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_standardised
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the standardised value of the CD for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the standardised value of the CD *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the standardised value of the CD are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the standardised CD value for each sample pair listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin_a, RangeIterator rend_a,
                       RangeIterator rbegin_b, RangeIterator rend_b)
```

**Description:** Returns the basic value of the CD measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of each sample are represented by a range of iterators; the species names of sample $A$ are represented by the range `[rbegin_a, rend_a)`, and the species names of $B$ are represented by the range `[rbegin_b, rend_b)`. Each iterator in these ranges refers to an object of type `std::string`. Every input string must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename_a, char* filename_b)
```

**Description:** Returns the basic value of the CD measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of sample $A$ are stored in file `filename_a`, and the species names of sample $B$ are stored in file `filename_b`. The names appear in each file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input files must match a species name in $\mathcal{T}$.

```
Number_type compute_expectation(int sample_size_a, int sample_size_b)
```

**Description:** Returns the expectation of the CD among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these pairs contributes with the same weight to the computation of the mean.

```
Number_type compute_variance(int sample_size_a, int sample_size_b)
```

**Description:** Returns the variance of the CD among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these pairs contributes with the same weight to the computation of the variance.

```
Number_type compute_deviation(int sample_size_a, int sample_size_b)
```

**Description:** Returns the standard deviation of the CD among all pairs of samples $A, B$ in the stored tree such that $A$ consists of `sample_size_a` leaves and $B$ consists of `sample_size_b` leaves. Each of these pairs contributes with the same weight to the computation of the deviation.

### 4.4.7 `Community_distance_nearest_taxon`

Unlike most measures, there are no moment functions available for the CDNT. Hence, there are also no functions available that compute the standardised value of this measure. Yet, we provide query functions for three different versions of the CDNT; the maximised, the averaged, and the directed CDNT–see Section 6.2 for the definitions of these measures.

```
Community_distance_nearest_taxon
(typename Community_distance_nearest_taxon::Tree_type &tree)
```

**Description:** Constructs an object of the `Community_distance_nearest_taxon` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_basic(char *matrix_filename_a,
                       char *matrix_filename_b,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the maximised CDNT for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair that consists of a sample in `matrix_filename_a` and a sample in `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_filename_a` consist of $n$ samples, and let `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic maximised CDNT values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic maximised CDNT values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_basic(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is an output iterator. The function computes the basic value of the maximised CDNT for every pair of samples in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair that consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic maximised CDNT values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the maximised CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the maximised CDNT measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the maximised CDNT measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic value of the maximised CDNT measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_basic(const std::vector<std::string> &names_a,
                   const std::vector<std::vector<bool> > &matrix_a,
                   const std::vector<std::string> &names_b,
                   const std::vector<std::vector<bool> > &matrix_b,
                   OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the maximised CDNT for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic maximised CDNT values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic maximised CDNT values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_basic(const std::vector<std::string> &names,
                       const std::vector<std::vector<bool> > &matrix,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the maximised CDNT for every pair of samples in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic maximised CDNT values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the maximised CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic value of the maximised CDNT measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the maximised CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the maximised CDNT measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic value of the maximised CDNT measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic maximised CDNT value for each sample pair that is listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_averaged_basic
(char *matrix_filename_a, char *matrix_filename_b, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the averaged CDNT for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair that consists of a sample in `matrix_filename_a` and a sample in `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_filename_a` consist of $n$ samples, and let `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic averaged CDNT values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic averaged CDNT values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_averaged_basic
(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is an output iterator. The function computes the basic value of the averaged CDNT for every pair of samples in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic averaged CDNT values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_averaged_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the averaged CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the averaged CDNT measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format described in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair that is listed in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_averaged_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the averaged CDNT measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic value of the averaged CDNT measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair that is listed in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_averaged_basic(const std::vector<std::string> &names_a,
                            const std::vector<std::vector<bool> > &matrix_a,
                            const std::vector<std::string> &names_b,
                            const std::vector<std::vector<bool> > &matrix_b,
                            OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the averaged CDNT for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic averaged CDNT values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic averaged CDNT values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_averaged_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the averaged CDNT for every pair of samples in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic averaged CDNT values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_averaged_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the averaged CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic value of the averaged CDNT measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_averaged_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries, OutputIterator ot)
```

**Description:** Computes the basic value of the averaged CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the averaged CDNT measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic value of the averaged CDNT measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic averaged CDNT value for each sample pair that is listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_directed_basic
(char *matrix_filename_a, char *matrix_filename_b,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the two basic values of the directed CDNT *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the two basic values of the directed CDNT for each of the $nm$ possible pairs that we get using a sample from each file.

The last two arguments of the function are output iterators. The function computes the two basic directed CDNT values for each sample pair that consists of a sample in `matrix_filename_a` and a sample in `matrix_filename_b`, and the results are returned through the output iterators. The computed values are returned in the following way; let `matrix_filename_a` consist of $n$ samples, and let `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through output iterator `ot_a_to_b` the function returns $nm$ values such that the $m$ first values are the basic directed CDNT values from sample $A_1$ to $B_1$, from $A_1$ to $B_2$, ..., from $A_1$ to $B_m$, the next $m$ values are the basic directed CDNT values from $A_2$ to $B_1$, from $A_2$ to $B_2$, ..., from $A_2$ to $B_m$, and so on. Through output iterator `ot_b_to_a` the function returns $nm$ values such that the $m$ first values are the basic directed CDNT values from sample $B_1$ to $A_1$, from $B_2$ to $A_1$, ..., from $B_m$ to $A_1$, the next $m$ values are the basic directed CDNT values from $B_1$ to $A_2$, from $B_2$ to $A_2$, ..., from $B_m$ to $A_2$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_directed_basic
(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the two basic values of the directed measure *for every pair* of samples that appear in `matrix_filename`. In particular, if file `matrix_filename` contains $n$ samples the function computes the two basic values of the directed CDNT measure for each of the $n^2$ possible pairs that we can create with these samples.

The second argument of the function is an output iterator. The function computes the basic value of the directed CDNT for every pair of query samples in `matrix_filename`, and the results are returned through the output iterator. The computed values are returned in the following way; let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic directed CDNT values from sample $A_1$ to $A_1$, from $A_1$ to $A_2$, ..., from $A_1$ to $A_n$, the next $n$ values are the basic directed CDNT values from $A_2$ to $A_1$, from $A_2$ to $A_2$, ..., from $A_2$ to $A_n$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_directed_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename_a` and in file `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the two basic values of the directed CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the directed CDNT measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last two arguments of the function are output iterators. The function computes the two basic directed CDNT values for each sample pair in `queries_filename`, and the results are returned through the output iterators. For every pair $(A, B)$ that appears in `queries_filename` such that $A$ is a sample in `matrix_filename_a` and $B$ is a sample in `matrix_filename_b`, iterator `ot_a_to_b` returns the basic value of the directed CDNT from $A$ to $B$, and iterator `ot_b_to_a` returns the basic value of the directed CDNT from $B$ to $A$. For every

output iterator, the order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_directed_specific_pairs_basic
(char *matrix_filename, char *queries_filename,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the two basic values of the directed CDNT measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic values of the directed CDNT measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last two arguments of the function are output iterators. The function computes the two basic directed CDNT values for each sample pair in `queries_filename`, and the results are returned through the output iterators. For every pair $(A_i, A_j)$ that appears in `queries_filename`, iterator `ot_a_to_b` returns the basic value of the directed CDNT from $A_i$ to $A_j$, and iterator `ot_b_to_a` returns the basic value of the directed CDNT from $A_j$ to $A_i$. For each output iterator, the order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_directed_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the two basic values of the directed CDNT *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the two basic values of the directed CDNT for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last two arguments of the function are output iterators. The function computes the two basic directed CDNT values for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterators. The computed values are

returned in the following way; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through output iterator `ot_a_to_b` the function returns $nm$ values such that the $m$ first values are the basic directed CDNT values from sample $A_1$ to $B_1$, from $A_1$ to $B_2$, ..., from $A_1$ to $B_m$, the next $m$ values are the basic directed CDNT values from $A_2$ to $B_1$, from $A_2$ to $B_2$, ..., from $A_2$ to $B_m$, and so on. Through output iterator `ot_b_to_a` the function returns $nm$ values such that the $m$ first values are the basic directed CDNT values from sample $B_1$ to $A_1$, from $B_2$ to $A_1$, ..., from $B_m$ to $A_1$, the next $m$ values are the basic directed CDNT values from $B_1$ to $A_2$, from $B_2$ to $A_2$, ..., from $B_m$ to $A_2$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_directed_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 OutputIterator ot)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the two basic values of the directed measure *for every pair* of samples in `matrix`. In particular, if `matrix` contains $n$ samples the function computes the basic value of the directed CDNT measure for each of the $n^2$ possible pairs that we can create with these samples.

The last argument of the function is an output iterator. The function computes the basic value of the directed CDNT for every pair of samples in `matrix`, and the results are returned through the output iterator. The computed values are returned in the following way; let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic directed CDNT values from sample $A_1$ to $A_1$, from $A_1$ to $A_2$, ..., from $A_1$ to $A_n$, the next $n$ values are the basic directed CDNT values from $A_2$ to $A_1$, from $A_2$ to $A_2$, ..., from $A_2$ to $A_n$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_directed_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors

should have the structure that is described in Section 4.5.2. The function computes the two basic values of the directed CDNT measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic values of the directed CDNT are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last two arguments of the function are output iterators. The function computes the two basic directed CDNT values for each sample pair in `queries`, and the results are returned through the output iterators. For every pair $(i, j)$ that appears in `queries`, iterator `ot_a_to_b` returns the basic value of the directed CDNT from the $i$-th sample in `matrix_a` to the $j$-th sample in `matrix_b`, and iterator `ot_b_to_a` returns the basic value of the directed CDNT from the $j$-th sample in `matrix_b` to the $i$-th sample in `matrix_a`. For every output iterator, the order of the returned results is the same as the order of pairs that appear in file `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_directed_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot_a_to_b, OutputIterator ot_b_to_a)
```

**Description:** Computes the basic values of the directed CDNT measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the two basic values of the directed CDNT measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the two basic values of the directed CDNT measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last two arguments of the function are output iterators. The function computes the basic directed CDNT value for each sample pair that is listed in `queries`, and the results are returned through the output iterators. For every pair $(i, j)$ that appears in `queries`, iterator `ot_a_to_b` returns the basic value of the directed CDNT from the $i$-th to the $j$-th sample in `matrix`, and iterator `ot_b_to_a` returns the basic value of the directed CDNT from the $j$-th to the $i$-th sample in `matrix`. For each output iterator, the order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin_a, RangeIterator rend_a,
                       RangeIterator rbegin_b, RangeIterator rend_b)
```

**Description:** Returns the basic value of the maximised CDNT measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of each sample are represented by a range of iterators; the species names of sample $A$ are represented by the range `[rbegin_a, rend_a)`, and the species names of $B$ are represented by the range `[rbegin_b, rend_b)`. Each iterator in these ranges refers to an object of type `std::string`. Every input string must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename_a, char* filename_b)
```

**Description:** Returns the basic value of the maximised CDNT measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of sample $A$ are stored in file `filename_a`, and the species names of sample $B$ are stored in file `filename_b`. The names appear in each file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input files must match a species name in $\mathcal{T}$.

### 4.4.8 Phylogenetic_Sorensens_similarity

Unlike most measures, there are no moment functions available for the PhyloSor. Hence, there are also no functions available that compute the standardised value of this measure.

```
Phylogenetic_Sorensens_similarity
(typename Phylogenetic_Sorensens_similarity::Tree_type &tree)
```

**Description:** Constructs an object of the `Phylogenetic_Sorensens_similarity` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_basic(char *matrix_filename_a,
                       char *matrix_filename_b,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the PhyloSor for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair that consists of a sample in file `matrix_filename_a` and a sample in file `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let the matrix in `matrix_filename_a` consist of $n$ samples, and let the matrix in `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic PhyloSor values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic PhyloSor values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_basic(char *matrix_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is

an output iterator. The function computes the basic value of the PhyloSor for every pair of samples in `matrix_filename`, and the results are returned through the output iterator. In particular, let `matrix_filename` consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic PhyloSor values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in the file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the PhyloSor measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. The pairs for which the function computes the basic value of the PhyloSor measure are described in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the PhyloSor measure *for specific pairs* of samples that appear in file `matrix_filename`. The pairs for which the function computes the basic value of the PhyloSor measure are listed in file `queries_filename`. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair in `queries_filename`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file `queries_filename`.

The return value of the function is an integer which is equal to the number of samples stored in file `matrix_filename`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_basic(const std::vector<std::string> &names_a,
                   const std::vector<std::vector<bool> > &matrix_a,
                   const std::vector<std::string> &names_b,
                   const std::vector<std::vector<bool> > &matrix_b,
                   OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the PhyloSor for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic PhyloSor values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic PhyloSor values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_basic(const std::vector<std::string> &names,
                       const std::vector<std::vector<bool> > &matrix,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the PhyloSor for every pair of samples that are described in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair that consists of the $i$-th and $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic PhyloSor values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic PhyloSor values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the PhyloSor measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic value of the PhyloSor measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the PhyloSor measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the PhyloSor measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic value of the PhyloSor measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic PhyloSor value for each sample pair listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin_a, RangeIterator rend_a,
                       RangeIterator rbegin_b, RangeIterator rend_b)
```

**Description:** Returns the basic value of the PhyloSor measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of each sample are represented by a range of iterators; the species names of sample $A$ are represented by the range `[rbegin_a, rend_a)`, and the species names of $B$ are represented by the range `[rbegin_b, rend_b)`. Each iterator in these ranges refers to an object of type `std::string`. Every input string must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename_a, char* filename_b)
```

**Description:** Returns the basic value of the PhyloSor measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of sample $A$ are stored in file `filename_a`, and the species names of sample $B$ are stored in file `filename_b`. The names appear in each file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input files must match a species name in $\mathcal{T}$.

### 4.4.9 `Unique_fraction`

Unlike most measures, there are no moment functions available for UniFrac. Hence, there are also no functions available that compute the standardised value of this measure.

---

```
Unique_fraction(typename Unique_fraction::Tree_type &tree)
```

---

**Description:** Constructs an object of the `Unique_fraction` class. The input tree is stored in the constructed object, and is used thereon by every member function for computing queries or moments of this measure.

---

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_basic(char *matrix_filename_a,
                       char *matrix_filename_b,
                       OutputIterator ot)
```

---

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files `matrix_filename_a` and `matrix_filename_b`. Files `matrix_filename_a` and `matrix_filename_b` should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in file `matrix_filename_a`, and $B$ is a sample in file `matrix_filename_b`. Therefore, if file `matrix_filename_a` contains $n$ samples and `matrix_filename_b` contains $m$ samples, the function computes the basic value of the UniFrac for each of the $nm$ possible pairs that we can create using a sample from each file.

The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair that consists of a sample in file `matrix_filename_a` and a sample in file `matrix_filename_b`, and the results are returned through the output iterator. The order of the returned results is the following; let the matrix in `matrix_filename_a` consist of $n$ samples, and let the matrix in `matrix_filename_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in file `matrix_filename_a` and $B_j$ is the $j$-th sample in `matrix_filename_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic UniFrac values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic UniFrac values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_filename_a`, and the second integer is equal to the number of samples in `matrix_filename_b`.

---

```
template<class OutputIterator>
int csv_matrix_query_basic(char *matrix_filename, OutputIterator ot)
```

---

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file `matrix_filename`. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the measure *for every pair* of samples in `matrix_filename`. The second argument of the function is an output iterator. The function computes the basic value of the UniFrac for every pair of samples

in matrix_filename, and the results are returned through the output iterator. In particular, let matrix_filename consist of $n$ samples, and let $(A_i, A_j)$ be the pair which consists of the $i$-th and the $j$-th sample in this file. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic UniFrac values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples stored in the file matrix_filename.

```
template<class OutputIterator>
std::pair<int, int>
csv_matrix_query_specific_pairs_basic
(char *matrix_filename_a, char *matrix_filename_b,
 char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in files matrix_filename_a and matrix_filename_b. Files matrix_filename_a and matrix_filename_b should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the UniFrac measure *for specific pairs* $(A, B)$ such that $A$ is a sample in file matrix_filename_a, and $B$ is a sample in file matrix_filename_b. The pairs for which the function computes the basic value of the UniFrac measure are described in file queries_filename. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair in queries_filename, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file queries_filename.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in matrix_filename_a, and the second integer is equal to the number of samples in matrix_filename_b.

```
template<class OutputIterator>
int csv_matrix_query_specific_pairs_basic
(char *matrix_filename, char *queries_filename, OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are listed in file matrix_filename. This file should be in the matrix file format that is described in Section 4.5.1. The function computes the basic value of the UniFrac measure *for specific pairs* of samples that appear in file matrix_filename. The pairs for which the function computes the basic value of the UniFrac measure are listed in file queries_filename. This is a text file that follows the sample pairs file format presented in Section 4.5.1. The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair in queries_filename, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in file queries_filename.

The return value of the function is an integer which is equal to the number of samples stored in file matrix_filename.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_basic(const std::vector<std::string> &names_a,
                   const std::vector<std::vector<bool> > &matrix_a,
                   const std::vector<std::string> &names_b,
                   const std::vector<std::vector<bool> > &matrix_b,
                   OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several samples of species in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. Therefore, if `matrix_a` contains $n$ samples and `matrix_b` contains $m$ samples, the function computes the basic value of the UniFrac for each of the $nm$ possible pairs that we can create using a sample from each matrix.

The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair that consists of a sample in `matrix_a` and a sample in `matrix_b`, and the results are returned through the output iterator. The order of the returned results is the following; let `matrix_a` consist of $n$ samples, and let `matrix_b` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `matrix_a` and $B_j$ is the $j$-th sample in `matrix_b`. Through the output iterator, the function returns $nm$ values such that the $m$ first values are the basic UniFrac values for pairs $(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_m)$, the next $m$ values are the basic UniFrac values for pairs $(A_2, B_1), (A_2, B_2), \ldots, (A_2, B_m)$, and so on.

The return value of the function is a pair of integers; the first integer is equal to the number of species samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_basic(const std::vector<std::string> &names,
                       const std::vector<std::vector<bool> > &matrix,
                       OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the measure *for every pair* of samples that appear in `matrix`. The last argument of the function is an output iterator. The function computes the basic value of the UniFrac for every pair of samples that are described in `matrix`, and the results are returned through the output iterator. In particular, let `matrix` consist of $n$ samples, and let $(A_i, A_j)$ be the pair that consists of the $i$-th and $j$-th sample in this matrix. Through the output iterator, the function returns $n^2$ values such that the $n$ first values are the basic UniFrac values for pairs $(A_1, A_1), (A_1, A_2), \ldots, (A_1, A_n)$, the next $n$ values are the basic UniFrac values for pairs $(A_2, A_1), (A_2, A_2), \ldots, (A_2, A_n)$, and so on.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template<class OutputIterator>
std::pair<int, int>
matrix_query_specific_pairs_basic
(const std::vector<std::string> &names_a,
 const std::vector<std::vector<bool> > &matrix_a,
 const std::vector<std::string> &names_b,
 const std::vector<std::vector<bool> > &matrix_b,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are provided in two matrices, one represented by vectors `matrix_a` and `names_a`, and the other represented by `matrix_b` and `names_b`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the UniFrac measure *for specific pairs* $(A, B)$ such that $A$ is a sample in `matrix_a`, and $B$ is a sample in `matrix_b`. The pairs for which the function computes the basic value of the UniFrac measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th sample in `matrix_a` and the $j$-th sample in `matrix_b`. The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in vector `queries`.

The return value of the function is a pair of integers; the first integer is equal to the number of samples in `matrix_a`, and the second integer is equal to the number of samples in `matrix_b`.

```
template<class OutputIterator>
int matrix_query_specific_pairs_basic
(const std::vector<std::string> &names,
 const std::vector<std::vector<bool> > &matrix,
 const std::vector<std::pair<int, int> > &queries,
 OutputIterator ot)
```

**Description:** Computes the basic value of the UniFrac measure for several pairs of species samples in the stored tree $\mathcal{T}$. The input samples are represented by vectors `matrix` and `names`. These vectors should have the structure that is described in Section 4.5.2. The function computes the basic value of the UniFrac measure *for specific pairs* of samples that appear in `matrix`. The pairs for which the function computes the basic value of the UniFrac measure are listed in vector `queries`. Each element of this vector is a pair of integers $(i, j)$ indicating the query pair that consists of the $i$-th and the $j$-th sample in `matrix`. The last argument of the function is an output iterator. The function computes the basic UniFrac value for each sample pair listed in `queries`, and the results are returned through the output iterator. The order of the returned results is the same as the order of pairs that appear in `queries`.

The return value of the function is an integer which is equal to the number of samples in `matrix`.

```
template <class RangeIterator>
Number_type list_query(RangeIterator rbegin_a, RangeIterator rend_a,
                       RangeIterator rbegin_b, RangeIterator rend_b)
```

**Description:** Returns the basic value of the UniFrac measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of each sample are represented by a range of iterators; the species names of sample $A$ are represented by the range `[rbegin_a, rend_a)`, and the species names of $B$ are represented by the range `[rbegin_b, rend_b)`. Each iterator in these ranges refers to an object of type `std::string`. Every input string must match a species name in $\mathcal{T}$.

```
Number_type list_query(char* filename_a, char* filename_b)
```

**Description:** Returns the basic value of the UniFrac measure for a given pair of species samples $A$ and $B$ in the stored tree $\mathcal{T}$. The species names of sample $A$ are stored in file `filename_a`, and the species names of sample $B$ are stored in file `filename_b`. The names appear in each file separated by commas and with no additional characters–see the description of the species names file format in Section 4.5.1 for an example. Each name in the input files must match a species name in $\mathcal{T}$.

## 4.5 The Format of Input Files and Matrices

In this section we present the format of files and matrix objects that are used as input arguments by many functions in the C++ package.

### 4.5.1 File Formats

**The Matrix File.** This is a standard `csv` file which stores a matrix. Each matrix column corresponds to a species in $\mathcal{T}$ (the tree stored by the measure object), and each row describes a species sample which we want to process. The only exception is the first row of the matrix (the column headers); every entry in this row stores the species name that corresponds to each column. The $i$-th sample contains the species of column $j$ if the $j$-th entry in the corresponding row has value one. Otherwise, this value should be zero. In fact, if the value of the matrix entry is any positive real number, then the corresponding species is considered as part of the sample. Negative values in the matrix entries are not accepted.

Ideally, the matrix should have as many columns as the number of leaves in $\mathcal{T}$, and each column header should match the name of a species in $\mathcal{T}$. A function has a normal execution even if the columns in the input matrix file correspond to a subset of the species in $\mathcal{T}$. However, an error is thrown if the matrix contains at least one column whose header does not correspond to a species in $\mathcal{T}$.

**The Species Names File.** This is a text file that stores a set of species names. The names appear in the file separated by commas, and with no additional characters. For example, a valid syntax for a file of this kind is the following:

```
Species_A, Species_B, Species_C, Species_D
```

Each name in the file must match a species name in the tree used by the measure object.

**The Sample Pairs File.** This is a text file which is used by two-sample measure functions. A file of this type describes which pairs of samples should be selected from two matrices (or from a single matrix) in order to perform queries. More specifically, let $A$ and $B$ be two different matrices and let $(A_i, B_j)$ indicate the pair which consists of the $i$-th sample in $A$ and the $j$-th sample in $B$. Let $M$ be a two-sample measure and suppose that we want to compute the (basic or standardised) value of this measure for the following pairs of samples: $(A_4, B_2)$, $(A_3, B_6)$, all pairs $(A_{16}, B_j)$ where $j$ ranges from 20 to 25, all pairs $(A_i, B_j)$ where $i$ ranges from 5 to 8 and $j$ ranges from 31 to 38, and all pairs $(A_i, B_{61})$ where $i$ ranges from 21 to 23. To describe these pairs in the sample pairs file, the following syntax should be used:

```
4/2,3/6,16/20-25,5-8/31-38,21-23/61
```

For instance, in the above snippet, the term `5-8/31-38` encodes all sample pairs $(A_i, B_j)$ where $i$ ranges from 5 to 8 and $j$ ranges from 31 to 38. Note that the first sample in a matrix is represented by number one, and not by zero. If a pair in this file consists of integers that are smaller than or equal to zero, or larger than the number of samples in a matrix, then an error is thrown. The sample pairs file should not contain any spaces. For member functions of two-sample measures that receive only one input matrix, it is implied that $A$ and $B$ are the same matrix.

### 4.5.2 A Matrix Representation for A Set of Samples

Instead of using `csv` files, some functions in the C++ package handle input samples in a different way. In these cases, the input samples are represented by two objects; an object `matrix` which is of type `std::vector<std::vector<bool> >` and an object `names` which is of type `std::vector<std::string>`. Vector `matrix` represents a two dimensional matrix of boolean values. Each column in the matrix corresponds to a species in $\mathcal{T}$, the tree used by the measure object. The name of the species that corresponds to the $i$-th column of `matrix` is provided in the $i$-th entry of vector `names`. Each row in `matrix` describes an input query sample. The $i$-th input sample contains the $j$-th species in `names` if entry `matrix[i,j]` has value `true`. Otherwise, the value of this entry should be `false`.

The number of columns in `matrix` must match the number of entries in `names`. Each entry in `names` must match a species name in $\mathcal{T}$. A query function has a normal execution even if the species names in `names` correspond to a subset of the species in $\mathcal{T}$. In any other case, the function throws an error.

## 4.6 The Interface of the Tree Classes

Each measure class `X` contains the type `X::Tree_type` which represents the phylogenetic tree objects handled by this measure. For two different measure classes `X` and `Y`, types `X::Tree_type` and `Y::Tree_type` may not be the same. This has to do with performance issues; to speed up the execution of query and moment functions, each measure class uses a tree representation that has a special internal structure. However, the interface of each tree class is the same among all tree types. Next we provide a detailed description of this common interface.

To construct a tree object, the user has to provide a text file, or a string of characters, that describes a phylogenetic tree in the standard Newick format. Each tree object has functions that allow the user to traverse the tree structure. Also, the tree object provides functions for finding if a specific string matches the name of a species in the tree, checking if the tree is ultrametric, computing the Sackin's index of the tree etcetera.

Each node in the tree is associated with a unique integer which we call the *index* of this node. For a tree that consists of $n$ nodes, the index values of the nodes belong to the interval $\{0, 1, \ldots, n-1\}$. Each node in the tree is represented by an object of the class `Tree_type::Node_type`. A `Node_type` object stores the indices of the parent and the children of this node in the tree. It also stores the taxon name of the node, and the branch length of the edge that connects this node with its parent. A tree object provides functions that allow the user to traverse the tree nodes based on their index values.

A tree object also provides the user with the opportunity to access all leaf nodes of the tree in a single scan. For this reason, the tree object maintains a range of iterators, where each iterator refers to a representation of a leaf node. The iterators are of the type `Tree_type::Leaves_iterator`.

Also, each tree object contains the type `Tree_type::Number_type` which is used for handling all numerical operations with this tree. This is the same as the number type provided in the kernel class.

We continue with presenting in detail the interface of the types `Tree_type::Node_type` and `Tree_type::Leaves_iterator`. Then we continue with the description of all member functions that are available for a tree object.

- `Node_type`: Represents a node in the tree. This class has the following members.
  - `std::string taxon`: The name of the species/taxon that corresponds to this node.
  - `double distance`: The branch length of the link that connects this node with its parent. This value is set to -1 for the root node.
  - `std::vector<int> children`: A vector storing the indices of the node's children. If the node is leaf, this vector is empty.
  - `int parent`: The index of the parent node in the tree.
  - `int number_of_children()`: The number of elements in vector `children`.
- `Leaves_iterator`: An iterator for accessing the leaves in the tree. The `*`-operator of this iterator returns an `std::pair`; the first element of this pair is an `std::string` with the name of the species that corresponds to this leaf. The second element in the pair is an integer which is equal to the index of the corresponding node in the tree.

Next we present the interface for all member functions of a tree object.

`Phylogenetic_tree_(void)`

**Description:** Constructs an empty tree object.

`void construct_from_string(std::string &tree_str)`

**Description:** Creates the tree structure from the input string `tree_str`. This string represents a phylogenetic tree in Newick format.

`void construct_from_file(const char *filename)`

**Description:** Creates the tree structure from the input file `filename`. This should be a text file which stores a phylogenetic tree in Newick format.

`Node_type& node(int i)`

**Description:** Returns the node of the tree that has index `i`.

`int number_of_nodes()`

**Description:** Returns the number of nodes in the tree.

`int number_of_leaves()`

**Description:** Returns the number of leaves in the tree.

`int root_index()`

**Description:** Returns the index of the root node of the tree.

`Node_type root()`

**Description:** Returns the root node of the tree.

`bool is_ultrametric()`

**Description:** Returns `true` if the tree is ultrametric, otherwise returns `false`. A tree is ultrametric if any two simple paths that connect the root with leaf nodes have the same cost. Yet, many available phylogenetic trees are not ultrametric in this sense, because some paths between the root and the leaves might have slightly different costs. These slight differences may come from numerical errors that occured during the construction of the tree dataset. For this reason, it is a standard practice to accept that a tree is ultrametric even if there is a small difference among the costs of such paths. Therefore, before calling this function, the user is advised to declare the maximum absolute difference that is allowed between two path costs so that the tree is still considered ultrametric. This can be done using the following function.

```
void set_is_ultrametric_predicate_precision
(Number_type &is_ultrametric_predicate_precision)
```

**Description:** Declares the maximum absolute difference that is allowed between the costs of paths that connect the root to the leaves, so that the tree is still considered ultrametric. By default, the value of this difference is set to 0.001.

```
Leaves_iterator leaves_begin()
```

**Description:** Returns an iterator to the first element in a range of objects that represent the tree leaves. Each object in this range is an `std::pair`; the first element of this pair is an `std::string` which stores the species name of a leaf. The second element is an integer which is equal to the index of the corresponding leaf node in the tree.

```
Leaves_iterator leaves_end()
```

**Description:** Returns an iterator to the past-the-end element in the range representing the tree leaves.

```
Leaves_iterator find_leaf(std::string &name)
```

**Description:** Checks if input string `name` matches the name of a species (leaf node) in the tree. If there is such a match, the function returns an iterator that points to a pair of elements; the first element is an `std::string` with the same value as `name`, and the second element is the index of the corresponding leaf node in the tree. If `name` does not match any species name in the tree, the function returns an iterator to the past-the-end element in the range representing the tree leaves; this is the same as the iterator returned by function `leaves_end()`.

```
Number_type sackins_index()
```

**Description:** Returns the Sackin's index of the tree. The Sackin's index is equal to the sum of the depths of all leaf nodes in the tree. The depth of a leaf is defined as the number of edges that appear on the path between the leaf and the root node.

```
template <class OUTSTRM>
void print_tree(OUTSTRM &outs)
```

**Description:** Prints the tree structure in text format through the output stream `outs`. The output stream `outs` can be of any type (e.g. file stream, `std::cout`), as soon as it provides the output operator `operator<<`.

```
void print_tree()
```

**Description:** Prints the tree structure in the standard output.

```
void clear()
```

**Description:** Discards the current structure of the tree, leaving a tree object with zero nodes.

## 4.7 Designing A Custom Numeric Traits Class

The C++ package allows the user to choose the number type that will be handled in the computations of the query and moment functions. In this way, the user can get higher precision results, in exchange for running time performance. There are several software packages that offer high precision number types. For example, the GNU Multiple Precision Arithmetic Library provides number types that can handle arbitrary precision computations between integers, rationals, and floating point numbers [7].

We can use any number type with the `PhyloMeasures` package, provided that we first design a simple class that "imports" this type to the kernel. This is the numeric traits class that we mentioned earlier, when describing the interface of the kernel. The numeric traits class should provide six types in its namespace; these are the number type and five functors (i.e. classes that do not store any data). Each of these functors handles a simple operation, like computing the absolute value of a number. More specifically, the numeric traits class should contain the following types:

- `Number_type`: The number type to be used for all numeric operations.
- `To_double`: This type should provide an operator with interface:

```
double operator()(Number_type a)
```

  The operator returns a `double` whose value is the closest possible to the input number `a`.

- `Ceiling`: This type should provide an operator with interface:

```
Number_type operator()(Number_type a)
```

  The operator returns a `Number_type` object whose value is equal to the smallest integer that is larger than the input number `a`.

- `Square_root`: This type should provide an operator with interface:

```
Number_type operator()(Number_type a)
```

  The operator returns a `Number_type` object that is equal/approximates the square root of the input number `a`. This functor is used only for computing the deviation of a phylogenetic measure. Hence, if `Number_type` supports exact additions, subtractions, multiplications and divisions, but does not support exact square root computations then all package functions produce exact results except the functions that compute either the deviation of a measure or its standardised value.

- `Absolute_value`: This type should provide an operator with interface:

```
Number_type operator()(Number_type a)
```

  The operator returns the absolute value of the input number `a`.

- `Is_exact`: This type should provide an operator with interface:

```
bool operator()(void)
```

  The operator should return `true` if `Number_type` can handle additions, subtractions, multiplications and divisions with arbitrary precision, otherwise returns `false`.

An example of a numeric traits class is available with the package source code, in the file `PhyloMeasures/Include/Numeric_traits_double.h`. This is a traits class which uses the C++ `double` type.

## 4.8 Command Line Programs

Together with the C++ code of `PhyloMeasures`, we provide several practical programs that can call all available functions in the package. These programs allow the user to process a set of queries, or compute the statistical moments of a measure with a single call from the command line. The programs are provided in the folder `PhyloMeasures/Programs` of the package. This folder also contains an example makefile for building the programs on a Linux-based environment. The makefile is provided only as a point of reference, and may not execute properly on every system.

Next we describe which functions are called by each program, and also which input parameters should be provided when calling a program from the command line.

### 4.8.1 `csv_query_PD`

Computes the basic or the standardised values of the unrooted PD for a set of species samples on a given tree. This program can be called in the following way:

```
csv_query_PD -tree TREE_FILE -csv CSV_FILE [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `CSV_FILE` is a `csv` file that stores a matrix with the input species samples. This file should be in the matrix file format that is described in Section 4.5.1. When the optional parameter `-standardised` is used, the program returns the standardised unrooted PD values for the input samples. Otherwise, the program returns the basic value of the unrooted PD for each of these samples.

### 4.8.2 csv_query_MPD

Computes the basic or the standardised values of the MPD for a set of species samples on a given tree. This program can be called in the following way:

```
csv_query_MPD -tree TREE_FILE -csv CSV_FILE [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `CSV_FILE` is a `csv` file that stores a matrix with the input species samples. This file should be in the matrix file format that is described in Section 4.5.1. When the optional parameter `-standardised` is used, the program returns the standardised values of the MPD for the input samples. Otherwise, the program returns the basic value of the MPD for each of these samples.

### 4.8.3 csv_query_MNTD

Computes the basic or the standardised values of the MNTD for a set of species samples on a given tree. This program can be called in the following way:

```
csv_query_MNTD -tree TREE_FILE -csv CSV_FILE [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `CSV_FILE` is a `csv` file that stores a matrix with the input species samples. This file should be in the matrix file format that is described in Section 4.5.1. When the optional parameter `-standardised` is used, the program returns the standardised values of the MNTD for the input samples. Otherwise, the program returns the basic value of the MNTD for each of these samples. **Note:** if parameter `-standardised` is provided and the input tree is not ultrametric, then the program does not produce any results. A tree is considered as ultrametric if, for any two simple paths between the root and a leaf node, the absolute difference between the two path costs is at most 0.001.

### 4.8.4 csv_query_CAC

Computes the basic or the standardised values of the CAC for a set of species samples on a given tree. This program can be called in the following way:

```
csv_query_CAC -tree TREE_FILE -csv CSV_FILE [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `CSV_FILE` is a `csv` file that stores a matrix with the input species samples. This file should be in the matrix file format that is described in Section 4.5.1. When the optional parameter `-standardised` is used, the program returns the standardised CAC values for the input samples. Otherwise, the program returns the basic value of the CAC for each of these samples.

### 4.8.5 csv_query_CBL

Computes the basic or the standardised value of the CBL for several pairs of species samples on a given tree. This program can be called in the following way:

```
csv_query_CBL -tree TREE_FILE -csv CSV_FILE_A [CSV_FILE_B]
[-specific_pairs PAIRS_FILE] [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameters `CSV_FILE_A` and `CSV_FILE_B` are `csv` files, each one storing a matrix with input species samples. These files should follow the matrix file format that is described in Section 4.5.1.

When the optional parameters `-specific_pairs PAIRS_FILE` are used, then the program returns a value for each pair of samples that are described in file `PAIRS_FILE`. This file should be in the sample pairs file format which is explained in Section 4.5.1. If these parameters are not provided, then the program returns a value for each sample pair that consists of a sample in `CSV_FILE_A` and a sample in `CSV_FILE_B` (if two `csv` files are given), or returns a value for each pair of samples in `CSV_FILE_A` (if only one `csv` file is given). When the optional parameter `-standardised` is used, the program returns the standardised values of the CBL for the described sample pairs. Otherwise, the program returns the basic value of the CBL for each of these pairs.

The results are returned in the following order; let `CSV_FILE_A` consist of $n$ samples, and let `CSV_FILE_B` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `CSV_FILE_A` and $B_j$ is the $j$-th sample in `CSV_FILE_B`. If two `csv` files are provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $nm$ values in the following way; the $m$ first values are the results for pairs $(A_1, B_1)$, $(A_1, B_2)$, ..., $(A_1, B_m)$, the next $m$ values are the results for pairs $(A_2, B_1)$, from $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on. If only file `CSV_FILE_A` is provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $n^2$ values in the following way; the $n$ first values are the results for pairs $(A_1, A_1)$, $(A_1, A_2)$, ..., $(A_1, A_n)$, the next $n$ values are the results for pairs $(A_2, A_1)$, from $(A_2, A_2)$, ..., $(A_2, A_n)$, and so on. If parameters `-specific_pairs PAIRS_FILE` are provided in the program call, the returned results have the same order as the pairs that appear in file `PAIRS_FILE`.

### 4.8.6 csv_query_CD

Computes the basic or the standardised value of the CD for several pairs of species samples on a given tree. This program can be called in the following way:

```
csv_query_CD -tree TREE_FILE -csv CSV_FILE_A [CSV_FILE_B]
[-specific_pairs PAIRS_FILE] [-standardised]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameters `CSV_FILE_A` and `CSV_FILE_B` are `csv` files, each one storing a matrix with input species samples. These files should follow the matrix file format that is described in Section 4.5.1.

When the optional parameters `-specific_pairs PAIRS_FILE` are used, then the program returns a value for each pair of samples that are described in file `PAIRS_FILE`. This file should be in the

sample pairs file format which is explained in Section 4.5.1. If this file is not provided, then the program returns a value for each sample pair that consists of a sample in `CSV_FILE_A` and a sample in `CSV_FILE_B` (if two `csv` files are given), or returns a value for each pair of samples in `CSV_FILE_A` (if only one `csv` file is given). When the optional parameter `-standardised` is used, the program returns the standardised values of the CD for the described sample pairs. Otherwise, the program returns the basic value of the CD for each of these pairs.

The results are returned in the following order; let `CSV_FILE_A` consist of $n$ samples, and let `CSV_FILE_B` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `CSV_FILE_A` and $B_j$ is the $j$-th sample in `CSV_FILE_B`. If two `csv` files are provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $nm$ values in the following way; the $m$ first values are the results for pairs $(A_1, B_1)$, $(A_1, B_2)$, ..., $(A_1, B_m)$, the next $m$ values are the results for pairs $(A_2, B_1)$, from $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on. If only file `CSV_FILE_A` is provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $n^2$ values in the following way; the $n$ first values are the results for pairs $(A_1, A_1)$, $(A_1, A_2)$, ..., $(A_1, A_n)$, the next $n$ values are the results for pairs $(A_2, A_1)$, from $(A_2, A_2)$, ..., $(A_2, A_n)$, and so on. If parameters `-specific_pairs PAIRS_FILE` are provided in the program call, the returned results have the same order as the pairs that appear in file `PAIRS_FILE`.

### 4.8.7 csv_query_CDNT

Computes the basic value of the CDNT for several pairs of species samples on a given tree. This program can be called in the following way:

```
csv_query_CDNT -tree TREE_FILE -csv CSV_FILE_A [CSV_FILE_B]
[-specific_pairs PAIRS_FILE] [ -maximised | -averaged | -directed ]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameters `CSV_FILE_A` and `CSV_FILE_B` are `csv` files, each one storing a matrix with input species samples. These files should follow the matrix file format that is described in Section 4.5.1.

When the optional parameters `-specific_pairs PAIRS_FILE` are used, then the program returns results for each pair of samples that are described in file `PAIRS_FILE`. This file should be in the sample pairs file format which is explained in Section 4.5.1. If this file is not provided, then the program returns results for each sample pair that consists of a sample in `CSV_FILE_A` and a sample in `CSV_FILE_B` (if two `csv` files are given), or returns results for each pair of samples in `CSV_FILE_A` (if only one `csv` file is given). When the optional parameter `-maximised` is used, the program returns the basic value of the maximised CDNT for each of the samples pairs. When the parameter `-averaged` is used, the program returns the basic value of the averaged CDNT for each of the samples pairs. When the parameter `-directed` is used, the program returns the two basic values of the directed CDNT for each of the samples pairs. At most one of these parameters should be used in a single call of the program. If none of the three is used, the program returns the basic value of the maximised CDNT for each of the sample pairs.

The results are returned in the following order; let `CSV_FILE_A` consist of $n$ samples, and let `CSV_FILE_B` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `CSV_FILE_A` and $B_j$ is the $j$-th sample in `CSV_FILE_B`. If two `csv` files are provided in the program call, and parameters `-directed` and `-specific_pairs` are absent, then the program

returns $nm$ values in the following way; the $m$ first values are the results for pairs $(A_1, B_1)$, $(A_1, B_2)$, ..., $(A_1, B_m)$, the next $m$ values are the results for pairs $(A_2, B_1)$, from $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on. If only file `CSV_FILE_A` is provided in the program call, and parameters `CSV_FILE_B` and `-specific_pairs` are absent, then the program returns $n^2$ values in the following way; the $n$ first values are the results for pairs $(A_1, A_1)$, $(A_1, A_2)$, ..., $(A_1, A_n)$, the next $n$ values are the results for pairs $(A_2, A_1)$, from $(A_2, A_2)$, ..., $(A_2, A_n)$, and so on. For this case, the results are returned likewise even if parameter `-directed` is provided in the program call. If parameters `-specific_pairs PAIRS_FILE` are provided in the program call, and parameter `-directed` is not, the returned results have the same order as the pairs that appear in file `PAIRS_FILE`.

If the optional parameter `-directed` is used in the program call, and two `csv` files are provided, then the basic values of the directed CDNT from samples in `CSV_FILE_A` to samples in `CSV_FILE_B` will be returned first (using the order described above), followed by the basic values of the directed CDNT from samples in `CSV_FILE_B` to samples in `CSV_FILE_A`.

### 4.8.8 csv_query_PhyloSor

Computes the basic value of PhyloSor for several pairs of species samples on a given tree. This program can be called in the following way:

```
csv_query_PhyloSor -tree TREE_FILE -csv CSV_FILE_A [CSV_FILE_B]
[-specific_pairs PAIRS_FILE]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameters `CSV_FILE_A` and `CSV_FILE_B` are `csv` files, each one storing a matrix with input species samples. These files should follow the matrix file format that is described in Section 4.5.1.

When the optional parameters `-specific_pairs PAIRS_FILE` are used, then the program returns results for each pair of samples that are described in file `PAIRS_FILE`. This file should be in the sample pairs file format which is explained in Section 4.5.1. If this file is not provided, then the program returns results for each sample pair that consists of a sample in `CSV_FILE_A` and a sample in `CSV_FILE_B` (if two `csv` files are given), or returns results for each pair of samples in `CSV_FILE_A` (if only one `csv` file is given).

The results are returned in the following order; let `CSV_FILE_A` consist of $n$ samples, and let `CSV_FILE_B` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `CSV_FILE_A` and $B_j$ is the $j$-th sample in `CSV_FILE_B`. If two `csv` files are provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $nm$ values in the following way; the $m$ first values are the results for pairs $(A_1, B_1)$, $(A_1, B_2)$, ..., $(A_1, B_m)$, the next $m$ values are the results for pairs $(A_2, B_1)$, from $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on. If only file `CSV_FILE_A` is provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $n^2$ values in the following way; the $n$ first values are the results for pairs $(A_1, A_1)$, $(A_1, A_2)$, ..., $(A_1, A_n)$, the next $n$ values are the results for pairs $(A_2, A_1)$, from $(A_2, A_2)$, ..., $(A_2, A_n)$, and so on. If parameters `-specific_pairs PAIRS_FILE` are provided in the program call, the returned results have the same order as the pairs that appear in file `PAIRS_FILE`.

### 4.8.9 csv_query_UniFrac

Computes the basic value of UniFrac for several pairs of species samples on a given tree. This program can be called in the following way:

```
csv_query_UniFrac -tree TREE_FILE -csv CSV_FILE_A [CSV_FILE_B]
[-specific_pairs PAIRS_FILE]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameters `CSV_FILE_A` and `CSV_FILE_B` are `csv` files, each one storing a matrix with input species samples. These files should follow the matrix file format that is described in Section 4.5.1.

When the optional parameters `-specific_pairs PAIRS_FILE` are used, then the program returns results for each pair of samples that are described in file `PAIRS_FILE`. This file should be in the sample pairs file format which is explained in Section 4.5.1. If this file is not provided, then the program returns results for each sample pair that consists of a sample in `CSV_FILE_A` and a sample in `CSV_FILE_B` (if two `csv` files are given), or returns results for each pair of samples in `CSV_FILE_A` (if only one `csv` file is given).

The results are returned in the following order; let `CSV_FILE_A` consist of $n$ samples, and let `CSV_FILE_B` consist of $m$ samples. Also, let $(A_i, B_j)$ indicate the pair of samples where $A_i$ is the $i$-th sample in `CSV_FILE_A` and $B_j$ is the $j$-th sample in `CSV_FILE_B`. If two `csv` files are provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $nm$ values in the following way; the $m$ first values are the results for pairs $(A_1, B_1)$, $(A_1, B_2)$, ..., $(A_1, B_m)$, the next $m$ values are the results for pairs $(A_2, B_1)$, from $(A_2, B_2)$, ..., $(A_2, B_m)$, and so on. If only file `CSV_FILE_A` is provided in the program call, and the `-specific_pairs` parameter is absent, then the program returns $n^2$ values in the following way; the $n$ first values are the results for pairs $(A_1, A_1)$, $(A_1, A_2)$, ..., $(A_1, A_n)$, the next $n$ values are the results for pairs $(A_2, A_1)$, from $(A_2, A_2)$, ..., $(A_2, A_n)$, and so on. If parameters `-specific_pairs PAIRS_FILE` are provided in the program call, the returned results have the same order as the pairs that appear in file `PAIRS_FILE`.

### 4.8.10 measure_moments

Computes the expectation (mean) and the deviation of a phylogenetic measure for a set of sample sizes on a given tree. The program can compute these two statistical moments for several measures supported in `PhyloMeasures`, except the CAC, CDNT, PhyloSor, and UniFrac. This program can be called in the following way:

```
measure_moments -tree TREE_FILE -measure MEASURE_NAME [-expectation | -deviation]
[-sample_size SIZE | -sample_size_a SIZE_A -sample_size_b SIZE_B |
 -sample_sizes_file SIZES_FILE]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `MEASURE_NAME` indicates the phylogenetic measure for which the program computes the expectation or the deviation. The accepted values for this parameter are: `PD`, `MPD`, `MNTD`, `CBL`, and `CD`. If this parameter is set to `MNTD` and the input tree is

not ultrametric, then the program does not produce any results. In this program, a tree is considered ultrametric if, for any two simple paths between the root and a leaf node, the absolute difference between the path costs is at most 0.001. Parameters `-expectation` and `-deviation` indicate the statistical moment that is computed by the program. Only one of these parameters can be provided in a single program call.

Parameters `-sample_size SIZE` indicate that the program computes the moment of a single-sample measure for sample size `SIZE`. These parameters can only be used when `MEASURE_NAME` is set either to `PD`, `MPD` or `MNTD`. Parameters `-sample_size_a SIZE_A -sample_size_b SIZE_B` indicate that the program computes the moment of a two-sample measure for sample sizes `SIZE_A` and `SIZE_B`. These parameters can only be used when `MEASURE_NAME` is set either to `CBL` or to `CD`.

If parameters `-sample_sizes_file SIZES_FILE` are provided, then the program computes the moment of the selected measure for all sample sizes that are listed in file `SIZES_FILE`. This is a text file whose syntax depends on whether a single-sample or a two-sample measure is selected. If `MEASURE_NAME` is set either to `PD`, `MPD` or `MNTD`, then `SIZES_FILE` should contain a set of non-negative integers (the sample sizes) separated by commas and with no spaces.

If `MEASURE_NAME` is set either to `CBL` or `CD` then `SIZES_FILE` should be structured as follows. Suppose that we want to compute the expectation of a two-sample measure for the next pairs of sizes: $(9, 12)$, $(21, 28)$, $(15, 10)$, and $(30, 12)$. To describe these pairs in file `SIZES_FILE`, we should use the following syntax:

```
9/12,21/28,15/10,30/12
```

The file should not contain any spaces.

#### 4.8.11 `CAC_moments`

Computes the centralised statistical moments of the CAC measure for a set of sample sizes on a given tree. This program can be called in the following way:

```
CAC_moments -tree TREE_FILE -chi VALUE_CHI -k VALUE_K
[-sample_size SIZE | -sample_sizes_file SIZES_FILE]
```

Parameter `TREE_FILE` is a text file that stores the input phylogenetic tree. The tree should be represented in the standard Newick format. Parameter `VALUE_CHI` indicates the value for the $\chi$ constant of the CAC measure–see the definition of the measure in Section 6 for more details. This value should belong to the interval $(0.5, 1]$. Parameter `VALUE_K` indicates that the program computes all first `VALUE_K` moments of the CAC measure. The first returned moment is the mean, and for $i$ larger than one, the $i$-th returned moment is the centralised moment of order $i$. Parameter `VALUE_K` must be a positive integer. Option `-sample_size SIZE` indicates that the program computes the moments of the CAC measure for sample size `SIZE`.

If parameters `-sample_sizes_file SIZES_FILE` are provided, then the program computes the moments of the CAC for all sample sizes that are described in file `SIZES_FILE`. This is a text file which contains a set of non-negative integers (the sample sizes) separated by commas.

# 5 Numeric Precision and Use of Custom Number Types

Standard software handles mathematical operations with numbers of limited precision. For instance, the C++ built-in `double` float type can represent accurately numbers that have up to 17 decimal digits. Due to this limited precision, basic mathematical operations are performed in an inexact manner. This introduces numerical errors, which may become larger and larger as the output of inexact operations is used for further calculations.

Numerical errors may occur also when computing the statistical moments of certain phylogenetic measures. The measures that are more sensitive to these errors are the PD, the MNTD, the CAC, and the CBL. This is because all known formulas for calculating the moments of these measures involve computations with binomial coefficients [16, 15]. Binomial coefficients are mathematical functions that are notorious for producing very large values. For example, for parameter values $n = 100$ and $r = 50$, the binomial coefficient $\binom{n}{r}$ is larger than $10^{30}$. More precisely, if both $n$ and $r$ are numbers represented by $x$ digits, in the worst case the value $\binom{n}{r}$ can have up to roughly $nx$ digits. Even for relatively small values of $n$ and $r$, standard fixed-precision arithmetic is not enough to represent these numbers. A similar problem arises when computing the higher order moments of a phylogenetic measure. High order moments may have very big values, which cannot be represented by fixed-precision numbers. This problem has been observed in `PhyloMeasures` when computing the higher order moments of the CAC measure.

In `PhyloMeasures`, we tackle these problems in two different ways. First, we implemented the moment functions of the phylogenetic measures so that binomial coefficients are not calculated explicitly; instead, we handle fractions of binomial coefficients, and we make sure that the value of any such fraction is not larger than one. However, the decimal part in these fractions may consist of many digits, and rounding errors occur in case the fractions are represented by fixed-precision numbers. In theory, this may induce a large numerical error in the calculated value of a statistical moment. In practice though, the difference between the exact value of the moment and the calculated one is very small. Later in this section we present experiments that prove this argument.

More than that, we designed the C++ version of the `PhyloMeasures` package so that it allows computations with exact arithmetic. As described in Section 4, in the C++ package the number type used by all functions is a parameter that can be selected by the user. In this way, the user can choose a number type among a large variety of existing libraries that implement high precision arithmetic [7, 5]. Of course, selecting a number type of higher precision will slow down the computations. It is up to the user to decide which is going to be the trade-off between precision and efficiency of the package functions. For the R version of the package, the number type used is the 64-bit double precision float.

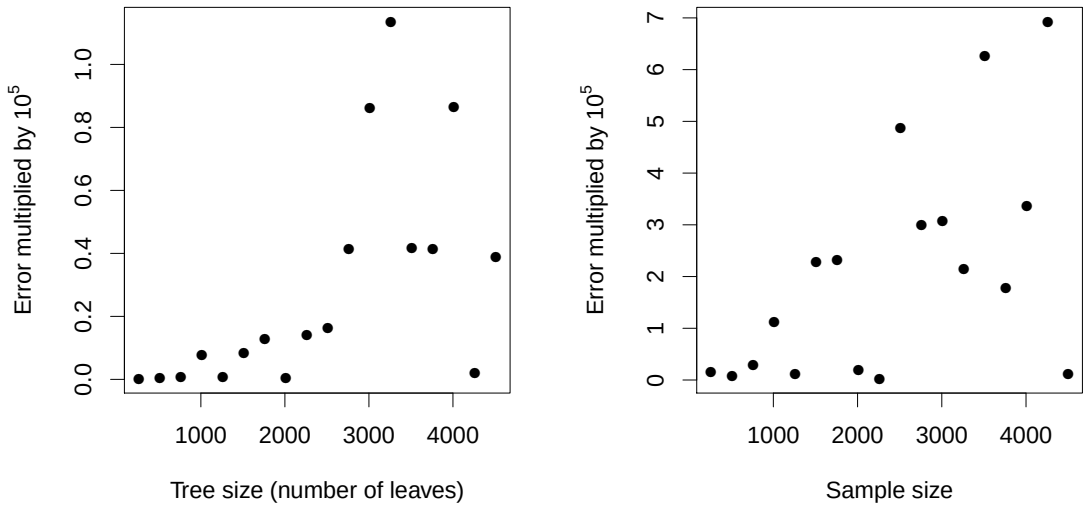
As we argued earlier, even when using standard floating point arithmetic, the numerical errors introduced during the computations of moment functions are quite small. To test this, we conducted experiments where we used both exact and inexact number types. In particular, we used the C++ version of our package, and we calculated the variance of the PD measure for various tree and sample sizes. For each tree and sample size we computed the value of the variance twice; the first time we used standard floating point arithmetic, and the second time we used an exact number type. The numerical error that occurs when using floating point arithmetic was calculated as the absolute difference between the two computed values.

For the floating point computations, we used double precision floats. For the exact computation of the variance, we used the GNU multiprecision quotient number type [7]. This type supports exact computations between rationals. All values that have a finite number of digits in the decimal system can be represented as rationals, and the edge lengths in phylogenetic tree data are fixed-precision

numbers; therefore, the GNU quotient type can compute exactly the expectation and the variance of any of the measures that we consider (but not the deviation since it involves a square root operation; the result of this operation is not always a rational, and in this case another number type should be used).

We performed two sets of experiments; in the first set we measured the error that appears when computing the variance for trees of different sizes and for a single sample size. In the second set, we considered a single tree and different sample sizes. For these experiments we used again the `mammals` phylogenetic tree that we describe in Section 2.3. Recall that this tree has 4,510 leaf nodes, and 6,618 nodes in total.

For the first set of experiments, we extracted again from `mammals` eighteen subtrees, each having $250k + 10$ leaves with $k$ ranging from one to eighteen. For each of these trees, we picked a sample size equal to half of the number of leaves in the tree, and we calculated the error in the PD variance for this tree and this sample size. Figure 4(a) illustrates the results of these experiments.




(a) (b)

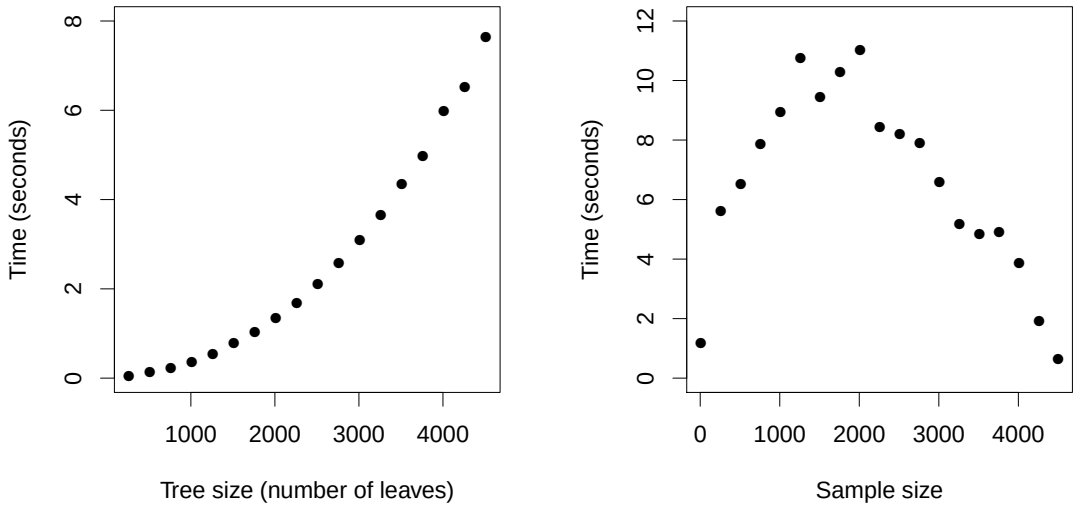
Figure 4: The results of experiments where we measured the absolute difference between the exact value of the PD variance and the value computed using double precision floats. (a) The absolute difference between the two values for trees of different size. (b) The absolute difference between the two values when the variance is computed for various sample sizes on a tree of 4,510 leaves.

We see that the numerical error produced by the use of double precision floats is very small; for all tree sizes that we considered the largest observed error value is less than $10^{-4}$. The numerical error seems to increase as the input tree size grows, yet this increase is not very large. We conducted similar experiments on larger phylogenetic trees, and we observed that even for trees of more than 50,000 leaves the error was not larger than $10^{-3}$.

In the second set of experiments, we measured how the error changes with respect to the sample size. For this we measured the error in the computation of the PD variance on the entire `mammals` tree, and for forty-five different sample sizes. The sample sizes that we chose are $s = 100k + 10$ with $k$ ranging from 1 to 44, and also sample size $s = 4{,}500$ (we did not choose a sample size equal to the total number of leaves because then even the program that uses inexact arithmetic can compute the

correct variance value, which is zero in this case). The results of these experiments are presented in Figure 4(b). It appears that, in general, the error in the result of the floating point computations grows as the sample size increases. The maximum error observed in these experiments is slightly smaller than $7 \cdot 10^{-5}$.

When conducting the above experiments, we also measured the running time of the function that calculates the PD variance using exact arithmetic. Figure 5 shows the results of these measurements.




(a) (b)

Figure 5: The execution time of the PD variance function in `PhyloMeasures` when using the GNU multiprecision quotient number type. (a) The execution time for trees of different size. (b) The execution time when the variance is computed for various sample sizes on a tree of 4,510 leaves.

When processing the entire `mammals` tree, the exact function takes close to twelve seconds to compute a single value. If we compare this with the running times that we got in Section 2.3, we see that the use of exact arithmetic affects the package performance a lot. The execution time of the exact function seems to depend also on the sample size; the execution time is larger for samples sizes which are close to half the number of leaves in the tree, and becomes smaller as we get to sample sizes which are closer either to zero, or the total number of leaves in the tree.

We performed more experiments, where we computed the error in the variance of other phylogenetic measures, and also using large trees generated by a randomised process. The results of these experiments are to a large extent similar with the ones that we describe above. We conclude that using fixed-precision arithmetic with the moment functions of `PhyloMeasures` does not produce significantly large numerical errors. For most practical applications, it seems that there is no need to use more precise (but also more inefficient) number types. In any case, the users can always choose to use more precise arithmetic with the C++ version of the package.

# 6 Appendix: Definitions of Phylogenetic Measures

The `PhyloMeasures` package provides functions for evaluating nine different phylogenetic biodiversity measures; four of these measures are single-sample measures, and the remaining five are two-sample ($\beta$-diversity) measures. In the rest of this appendix we provide a formal definition for each of these measures.

Let $\mathcal{T}$ be a phylogenetic tree, and let $S$ be a sample (subset) of its leaves that consists of $s = |S|$ elements. We use $E$ to denote the set of edges in $\mathcal{T}$, and for every edge $e \in E$ we use weight$(e)$ to denote the weight of this edge. We call the *cost* of a path, or a subtree, in $\mathcal{T}$ the sum of the weights of the edges that form this path or subtree.

## 6.1 Single-Sample Measures

**Phylogenetic Diversity.** The *Phylogenetic Diversity* (PD) of sample $S$ is equal to the cost of the minimum-size subtree of $\mathcal{T}$ that connects all the nodes in $S$ [4]. An edge $e$ belongs to this subtree if $e$ appears on at least one simple path between a pair of leaves in $S$. More formally, the PD of sample $S$ is equal to:

$$\mathrm{PD}(\mathcal{T}, S) = \sum_{e \in E(S)} \mathrm{weight}(e),$$

where $E(S)$ is the set of all edges that appear on a simple path between a pair of leaves in $S$. This definition of the PD is known in the related literature as the *unrooted* PD.

**Mean Pairwise Distance.** The *Mean Pairwise Distance* (MPD) of sample $S$ is equal to the average cost of a path between any two distinct leaves in $S$ [17]. Therefore, the MPD of $S$ is equal to:

$$\mathrm{MPD}(\mathcal{T}, S) = \frac{2}{s(s-1)} \sum_{\{u,v\} \in S} cost(u, v) \ .$$

**Mean Nearest Taxon Distance.** The *Mean Nearest Taxon Distance* (MNTD) of sample $S$ is equal to the average path cost between an element in $S$ and its closest neighbour in $S$ [18]. More formally, the MNTD of $S$ is equal to:

$$\mathrm{MNTD}(\mathcal{T}, S) = \frac{1}{s} \sum_{v \in S} \min_{u \in S/\{v\}} cost(u, v) \ .$$

**Core Ancestor Cost.** Let $\chi$ be any real in the interval $(0.5, 1]$. We use $v_{\mathrm{anc}}(S, \chi)$ to denote the deepest node in $\mathcal{T}$ that has at least $\chi \cdot s$ elements of $S$ in its subtree. We call this node the *core ancestor of $S$ given $\chi$*. We call the cost of the simple path that connects $v_{\mathrm{anc}}(S, \chi)$ with the root of $\mathcal{T}$ the *Core Ancestor Cost of $S$ given $\chi$*, and we denote this cost by $\mathrm{CAC}(\mathcal{T}, S, \chi)$. We introduced the CAC in a previous paper considering that it can be a useful tool for phylogenetic analyses [15]. This measure can be used to test if a leaf sample $S$ consists mainly of a single group of closely related species, or $S$ is made of several unrelated clusters of species. For example, suppose that $\mathrm{CAC}(\mathcal{T}, S, 0.8)$ is relatively large, and comparable to the average path cost between the root and any leaf in $\mathcal{T}$. This means that at least 80% of the leaves in $S$ have a common ancestor which is deep in the tree, and they are closely

related. However, if $\mathrm{CAC}(\mathcal{T}, S, 0.51)$ is zero then $S$ is made of at least two main unrelated clusters of species.

### 6.2 Two-Sample Measures

We continue with the definition of the two-sample measures that are provided in `PhyloMeasures`. Let $\mathcal{T}$ be a phylogenetic tree, and let $A$ and $B$ be two samples (subsets) of leaves in $\mathcal{T}$ , such that $A$ has $a = |A|$ leaves, and $B$ has $b = |B|$ leaves. These two samples may not necessarily have the same size, and may not necessarily share any common elements.

**Common Branch Length.** The *Common Branch Length* (CBL) between samples $A$ and $B$ is defined as follows:

$$\mathrm{CBL}(\mathcal{T}, A, B) = \sum_{e \in E(A) \cap E(B)} \mathrm{weight}(e) \ ,$$

where $E(A)$ and $E(B)$ are respectively the set of all edges that appear on a simple path between a pair of leaves in $A$, and the set of all edges that appear on a simple path between leaves in $B$. Hence, the common branch length between $A$ and $B$ is equal to the sum of the weights of all those tree edges that fall on at least one path that connects two tips in $A$, and on at least one path between two tips in $B$ [6, 13]. The CBL is an extension of the PD measure for computing a diversity value between two samples of leaves.

**Community Distance.** The *Community Distance* (CD) between samples $A$ and $B$ is equal to:

$$\mathrm{CD}(\mathcal{T}, A, B) = \frac{1}{ab} \sum_{u \in A} \sum_{v \in B} cost(u, v) \ .$$

Hence, the community distance between the two samples is equal to the average cost of all paths that connect a leaf in $A$ with a leaf in $B$ [6, 13]. This measure is an extension of the MPD for calculating a diversity value between two samples of species.

**Community Distance Nearest Taxon.** The *Community Distance Nearest Taxon* (CDNT) *from* sample $A$ *to* sample $B$ is equal to the average distance between an element in $A$ and its closest neighbour in $B$ [6, 13]. That means:

$$\mathrm{CDNT}(\mathcal{T}, A, B) = \frac{1}{a} \sum_{u \in A} \min_{v \in B} cost(u, v) \ .$$

Unlike the previous two-sample measures, the CDNT is not symmetric. That means $\mathrm{CDNT}(\mathcal{T}, A, B)$ is not always equal to $\mathrm{CDNT}(\mathcal{T}, B, A)$. We also refer to $\mathrm{CDNT}(\mathcal{T}, A, B)$ as the *directed* CDNT from $A$ to $B$. We call the following measure the *averaged* CDNT between $A$ and $B$:

$$\mathrm{CDNT}_{\mathrm{avg}}(\mathcal{T}, A, B) = \frac{\sum_{u \in A} \min_{v \in B} cost(u, v) + \sum_{u \in B} \min_{v \in A} cost(u, v)}{a + b} \ .$$

We call the following measure the *maximised* CDNT between $A$ and $B$:

$$\mathrm{CDNT}_{\mathrm{max}}(\mathcal{T}, A, B) = \max(\mathrm{CDNT}(\mathcal{T}, A, B), \mathrm{CDNT}(\mathcal{T}, B, A)) \ .$$

The averaged and the maximised version of the CDNT are symmetric functions.

**Phylogenetic Sorensen's Similarity.** The *Phylogenetic Sorensen's Similarity* (PhyloSor) between samples $A$ and $B$ is equal to:

$$\text{PhyloSor}(\mathcal{T}, A, B) = \frac{2 \cdot \text{CBL}(\mathcal{T}, A, B)}{\text{PD}(\mathcal{T}, A) + \text{PD}(\mathcal{T}, B)} \ .$$

Therefore, the phylogenetic Sorensen's similarity is equal to two times the common branch length between the two samples, divided by the sum of the PD values for these two sets [6, 13].

**Unique Fraction.** Let $\mathcal{T}(A \cup B)$ denote the smallest subtree of $\mathcal{T}$ that contains all tips in $A \cup B$. Note that the total cost of $\mathcal{T}(A \cup B)$ (the sum of its branch lengths) is equal to $\text{PD}(A \cup B)$, the unrooted PD of the union of these sets. Let $E(A, B)$ denote the edges in $\mathcal{T}(A \cup B)$ such that for every $e \in E(A, B)$ the subtree that extends below $e$ contains at least one tip that appears in $A$, and at least one tip that appears in $B$. The *Unique Fraction* (UniFrac) between samples $A$ and $B$ is equal to:

$$\text{UniFrac}(\mathcal{T}, A, B) = \frac{\text{PD}(A \cup B) - \sum_{e \in E(A,B)} \text{weight}(e)}{\text{PD}(A \cup B)} \ .$$

In other words, the Unique Fraction is a ratio of two values. The numerator is equal to the sum of the branch lengths for the edges in $\mathcal{T}(A \cup B)$ whose subtrees contain either only tips of $A$, or only tips of $B$. The denominator is the total branch length of $\mathcal{T}(A \cup B)$ [9].

# References


[1] O.R.P. Bininda-Emonds, M. Cardillo, K.E. Jones, R.D.E MacPhee, R.M.D. Beck, R. Grenyer, S.A. Price, R.A. Vos, J.L. Gittleman and A. Purvis. The Delayed Rise of Present-Day Mammals. *Nature* 446: 507–512, 2007.

[2] M.G.B. Blum and O. François. On Statistical Tests of Phylogenetic Tree Imbalance: The Sackin and Other Indices Revisited. *Mathematical Biosciences*, 195:14–153, 2005.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, MIT press, Cambridge Masachussets, 2001.

[4] D.P. Faith. Conservation Evaluation and Phylogenetic Diversity. *Biological Conservation*, 61: 1–10, 1992.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2):13, 2007.

[6] C.H. Graham and P.V.A. Fine. Phylogenetic Beta Diversity: Linking Ecological and Evolutionary Processes Across Space and Time. *Ecology Letters*, 11: 1265:1277, 2008.

[7] T. Granlund. The GNU Multiple Precision Arithmetic Library. *TMG Datakonsult*, 2(2), Boston, 1996.

[8] S.W. Kembel, D.D. Ackerly, S.P. Blomberg, W.K. Cornwell, P.D. Cowan, M.R. Helmus, H. Morlon and C.O. Webb. *Documentation for Picante R Package*, 2011.

[9] C. Lozupone and R. Knight. UniFrac: a New Phylogenetic Method for Comparing Microbial Communities. *Applied and Environmental Microbiology*,71(12):8228-35, 2005.

[10] D.A. Nipperess. Phylodiv: an R Function for Calculating the Phylogenetic Diversity of Ecological Samples.

`http://davidnipperess.blogspot.dk/2012/07/heres-function-i-have-written-for-r.html`.

[11] D.A. Nipperess and F.A. Matsen IV. The Mean and Variance of Phylogenetic Diversity Under Rarefaction. *Methods in Ecology and Evolution*, 4: 566–572, 2013.

[12] E. Paradis, J. Claude and K. Strimmer. APE: Analyses of Phylogenetics and Evolution in R Language. *Bioinformatics*, 20: 289–290, 2004.

[13] N.G. Swenson. Phylogenetic Beta Diversity Metrics, Trait Evolution and Inferring Functional Beta Diversity of Communities. *PLoS ONE*: 6: e21264, 2011.

[14] C. Tsirogiannis and B. Sandel. Computing the Skewness of the Phylogenetic Mean Pairwise Distance in Linear Time. *Algorithms for Molecular Biology*, 9:15, 2014.

[15] C. Tsirogiannis, B. Sandel and Adrija Kalvisa. New Algorithms for Computing Phylogenetic Biodiversity. *Algorithms in Bioinformatics*, LNCS 8701:187–203, 2014.

[16] C. Tsirogiannis, B. Sandel and D. Cheliotis. Efficient Computation of Popular Phylogenetic Tree Measures. *Algorithms in Bioinformatics*, LNCS 7534:30–43., 2012.

[17] C.O. Webb. Exploring the Phylogenetic Structure of Ecological Communities: An Example for Rain Forest Trees. *The American Naturalist*, 156: 145–155, 2000.

[18] C.O. Webb, D.D. Ackerly, M.A. McPeek and M.J. Donoghue. Phylogenies and Community Ecology. *Annual Review of Ecology and Systematics* 33: 475–505, 2002.